(2)

CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

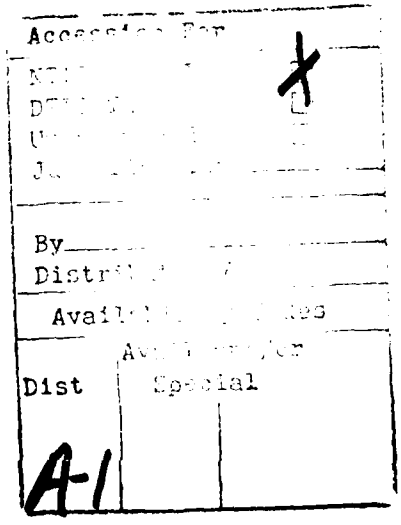
Subject: **Final Report - An Approach to Tailoring the Ada Runtime Environment**

CIN: C02 092LA 0004

15 FEBRUARY 1989

FINAL REPORT

AN APPROACH TO TAILORING THE ADA RUNTIME ENVIRONMENT



CONTRACT NUMBER: MDA903-87-D-0056

IITRI PROJECT NUMBER: T06168

PREPARED FOR:

U.S. ARMY, CECOM
ADVANCED SOFTWARE TECHNOLOGY
AMSEL-RD-SE-AST-SS-R
FT. MONMOUTH, N.J. 07703-5000

PREPARED BY:

IIT RESEARCH INSTITUTE
4600 FORBES BLVD.
LANHAM, MD 20706

DECEMBER 1988

TABLE OF CONTENTS

		PAGE
1.0	INTRODUCTION	1
	1.1 Background	1
	1.2 Project Scope	2
	1.3 Report Organization	3
2.0	WHAT IS TAILORING	4
	2.1 Essentials of Tailoring	4
	2.1.1 Definition of Tailoring	4
	2.1.2 Relationship to Configuring	4
	2.2 The Runtime Environment	5
	2.2.1 Definition of the Runtime Environment	5
	2.2.2 Relationships Between the Ada Compilation System and the Runtime Environment	7
	2.3 Overview of Tailorability	9
	2.3.1 Candidates for Tailoring	9
	2.3.2 Methods of Tailoring	12
3.0	HOW CAN ONE DETERMINE WHETHER AN RTE CAN BE TAILORED	15
	3.1 Criteria Indicating Design Features which Promote RTE Tailorability	15
	3.1.1 Clarity	16
	3.1.2 Concision	17
	3.1.3 Consistency	18
	3.1.4 Generality	18
	3.1.5 Modularity	28
	3.1.6 Expandability	20
	3.1.7 Self-Documentation	20
	3.1.8 Simplicity	21
	3.2 Criteria Indicating Services which Promote RTE Tailorability	21
	3.2.1 Availability of Source for the RTL	22
	3.2.2 Availability of Vendor Support	22
	3.2.3 RTL Source Language	22
	3.2.4 Sufficient Functionality of the RTL	22
	3.2.5 Documentation of Features that Support Tailorability	23
	3.2.6 Training	23
	3.2.7 Update Services	23
	3.3 Tailorability Checklist	24
4.0	WHAT CAN ONE DO TO TAILOR AN RTE	27
	4.1 Opportunities for Tailoring within the RTE	27
	4.1.1 Task Activation	27
	4.1.2 Task Scheduling	28
	4.1.3 Interrupts	29
	4.1.4 Storage Allocation	32
	4.1.5 Exceptions	34
	4.1.6 Input/Output	35
	4.1.7 Target Specific Functions	37

4.2 General Steps Involved in Tailoring 38
4.3 Examples of Tailoring 39
4.3.1 Device Driver 39
4.3.2 Runtime Memory Layout 53
4.3.3 Clock Correlation 56
4.3.4 Processor Scheduling 57

5.0 WHAT ARE THE POSSIBLE SIDE EFFECTS OF TAILORING 58
5.1 Factors to be Aware Of 58
5.1.1 The Interface 58
5.1.2 Inline Code 59
5.1.3 Hardware Considerations 59
5.2 Recovering from Errors Incurred by Tailoring 61

6.0 WHAT ARE THE COSTS OF TAILORING 64
6.1 Cost Impacts of Tailoring on Embedded Systems Development 64
6.1.1 Costs Associated with Responsibility 64
6.1.2 Costs Associated with Expertise 64
6.1.3 Costs Associated with the Development Cycle 64
6.2 Automation of the Tailoring Process 66

7.0 WHAT CAN VENDORS DO TO MAKE RTES MORE TAILORABLE 70
7.1 The Need for RTE Tailorability 70
7.2 Modularity and Commonality as a Basis for Tailorability 71
7.2.1 Modularity 72
7.2.2 Commonality 73
7.3 Benefits for Vendors 74

8.0 CONCLUSION 76

9.0 BIBLIOGRAPHY 78

APPENDIX A: TECHNICAL DISCUSSION OF THE RTE A-1
A.1 Task Identification A-2
A.2 Task Dependence A-3
A.3 Task Activation A-3
A.4 Task Scheduling A-4
A.5 Task Synchronization A-5
A.6 Task Termination A-7
A.7 Priorities A-8
A.8 Interrupts A-9
A.9 Timing Services A-11
A.10 Storage Allocation A-12
A.11 Exceptions A-13
A.12 Input/Output A-14

LIST OF FIGURES

FIGURE		PAGE
1	How an Ada Compilation System Produces an Executable Program	8
2	Candidates for Tailoring	10
3	Ways to Tailor an RTL	13
4	Tailorability Checklist	25
5	Application of RTE Development Tools	69

1.0 INTRODUCTION

This document reports on findings of the task titled "Develop an Approach to Tailoring the Ada Runtime Environment."

1.1 BACKGROUND

Implementors of real-time embedded systems are typically faced with integrating complex, multi-tasking software with target systems that provide limited physical space and thus limited computing facilities. In addition, these systems must often respond to real world events within very short time frames.

The developers' task is further complicated by the fact that the services of a standard executive program are rarely available for real-time embedded environments. As a result, functionality that is typically provided by such an executive in general purpose systems has to be designed and coded independently as part of the application. This demands a deep understanding of the architecture of the target computer and precise and efficient programming techniques. In the past, such software has usually been coded in assembly language and uniquely fitted to the target architecture and application.

The Department of Defense (DoD) now requires that such programs be coded in the DoD High Order Language Ada. Ada provides many advantages. It minimizes the coding effort by providing high level constructs that support multi-tasking and other real-time requirements. It supports good software design principles, and it provides a vehicle for reusable and portable software.

Difficulties have been encountered, however, in the use of Ada for real-time embedded systems. These difficulties stem mainly from the fact that the two outputs of an Ada compilation system, the translated source code and the runtime environment (RTE), often are too large, perform

inefficiently, or do not provide any special functionality for real-time embedded systems.

Ada compilation systems are likely to evolve and improve but, in the near term, implementors of embedded systems are finding that they often have to tailor the RTE to achieve their goals.

1.2 PROJECT SCOPE

This study is an investigation of RTE tailoring. It attempts to answer the following questions:

- What is tailoring?
- How can one determine whether an RTE can be tailored?
- What can one do to tailor an RTE?
- What are the possible side effects of tailoring?
- What are the costs of tailoring?
- What can vendors do to make RTEs more tailorable?

To a limited extent, the performance of Ada programs can be improved by techniques other than changing the code of the RTE. (see Section 2.3.1). This research, however, specifically addresses tailoring the code of the RTE to improve the performance or functionality of Ada executables. This effort is also targeted at real-time systems that are to be embedded in bare computers where both the application and executive services are implemented through Ada constructs and the RTE.

This research is not intended to establish direction for changes to a future Ada standard. It is intended to suggest near-term software engineering practices that conform to the Ada standard and can be applied when it is necessary to tailor the Ada RTE for highly constrained applications. It is also hoped that vendors will use this research to provide design features in Ada compilers that make it straightforward to employ these practices.

1.3 REPORT ORGANIZATION

Section 1.0: The present section is an introduction to the report.

Section 2.0: What is tailoring? In this section we attempt to clarify the concept of tailoring and to distinguish it from the related concept of configuring. Because the definition of tailoring depends on the notion of an RTE, we also discuss the RTE, describing its relationship to the Ada compilation system and the runtime library. We conclude with an analysis of which parts of the RTE are tailorable and how they can be tailored.

Section 3.0: How can one determine whether an RTE can be tailored? In this section we suggest criteria for determining the relative tailorability of the RTE for a given compilation system and criteria for evaluating the services provided by the vendor to the user.

Section 4.0: What can one do to tailor an RTE? In this section we examine which features of the RTE are candidates for tailoring. The section concludes with a summary of steps one would follow to tailor an RTE, and a set of examples.

Section 5.0: What are the possible side effects of tailoring? In this section we discuss other factors involved in tailoring the RTE, with special emphasis on the possible effects of errors introduced by tailoring the RTE.

Section 6.0: What are the costs of tailoring? In this section we explore the costs that might be entailed when tailoring the RTE. Topics include cost/benefit factors involved in tailoring the RTE, and cost-effective ways to automate the tailoring process.

Section 7.0: What can vendors do to make RTEs more tailorable? In this section we examine ways vendors can provide better RTE tailorability and ways they might be influenced to do so.

Section 8.0: We provide a summary of our results.

2.0 WHAT IS TAILORING

2.1 ESSENTIALS OF TAILORING

2.1.1 Definition of Tailoring

For the purposes of this study, tailoring may be defined as follows:

Tailoring: making changes to the code of an RTE in order to improve performance, improve utilization of computing resources, or implement functionality not obtainable through a particular version of Ada RTE.

This definition of "tailoring" should not encompass the alteration of a software system to correct a fault or to make massive changes to the point that the original system is no longer recognizable. The key idea is that the RTE to be tailored is logically correct but needs to be adapted, by modification of its code, for a new or constrained operating environment. For the purposes of this definition, the operating environment consists of the target computer and the requirements of the application.

2.1.2 Relationship to Configuring

Although this study does not address configurability of RTEs, the literature examined for this study often uses the terms "tailorability" and "configurability" together or in a similar context, and it is important to clarify their relationship. Configurability involves no code changes but is a feature that allows selective inclusion, through some mechanism, of modules or subroutines into the RTE. Configurability may be automatic and invisible, and determined by the compiler, semi-automatic through pragmas or linker directives, or highly visible and manual through explicit instructions to the linking process.

To illustrate the difference between tailoring and configuring, an equivalent definition of configuring is proposed:

Configuring: selectively including functional units of code in an RTE in order to improve performance, improve utilization of computing resources, or implement functionality not obtainable through a particular version of Ada RTE.

Configurability of the RTE is a valuable asset of an Ada compilation system. In many ways such a compiler feature is intended to minimize the need to tailor the RTE; i.e., it may not be necessary to tailor an RTE if sufficient configurability is provided. However, it is unlikely that a configurable design will make it possible to entirely avoid the need to tailor Ada RTEs in the immediate future.

Baker (13) suggests the following analogy for distinguishing between tailoring and configuring:

> "Let us call the actual modification of the code of the Ada compilation system 'tailoring', because it is analogous to the kind of alteration a tailor does to fit a ready made suit to a customer, which involves cutting and stitching. Let us contrast this with a less extreme form of adjustment, which we call 'configuration' in which the user chooses options and supplies parameters within a scheme set up by the compilation system. (This is analogous to choosing the best-fitting jacket and the best-fitting pants from two racks containing different sizes and styles, or using a belt to adjust the fit of clothing that is too loose, so that no alteration is necessary.)"

2.2 THE RUNTIME ENVIRONMENT

2.2.1 Definition of the Runtime Environment

The definition of the RTE used in this study is adopted from "A Framework for Describing Ada Runtime Environments" published by the Ada Runtime Environment Working Group of SIGAda (ARTEWG) (25). This definition is as follows:

Runtime Environment - set of all capabilities provided by three basic elements: predefined subroutines, abstract data conventions, and control structure code conventions."

This definition of the Ada RTE requires three subsidiary definitions. Predefined subroutines are pre-coded subroutines that are included in a program, usually implicitly, to implement standard conventions for accessing the facilities of the target computer. These subroutines are likely to be selected by the compilation system from a library of such routines called the Runtime Library, or RTL. Abstract data conventions are common conventions such as records, arrays, stacks, etc., for organizing and manipulating data. Control structure code conventions consist of in-line code that implements common conventions for handling control and data structures. In conventional bare machine programming these conventions may be included in software simply via enforced coding practices or through the use of predefined macros. In Ada or other high-level languages they are likely to be included through at least a one-level translation of high-level language constructs.

Of particular interest to this study are implementations for bare machines. These are machines in which there is no pre-existing operating system or executive, and in which functions that are typically the responsibility of an executive are likely to be provided by the compiler as part of the RTE. Each RTE downloaded onto a bare machine is tightly coupled with the application and loaded with it. The RTE may be unique to the application if the compilation system can selectively build the RTE by including only functionality that is needed. This is in contrast to the standard resident executives used by non-embedded systems, which are the same for all applications. The RTE is essentially an executive that is generated by an Ada compilation system for a specific application. The RTE does not include object code that results from a direct translation of an Ada programmer's logical implementation.

The RTE and the bare target machine provide a virtual or logical machine on which the Ada program can execute (8). The RTE provides lower

level functions that are not visible to the Ada programmer and not directly provided by the target architecture.

2.2.2 Relationships Between the Ada Compilation System and the Runtime Environment

The Ada RTE and the Ada code translations are generated by an Ada compilation system. An Ada compilation system is defined by the ARTEWG Framework as follows:

> Ada Compilation System - All the elements necessary to translate an Ada application program into an executable program; this usually includes the compiler, the linker, and the runtime library."

To clarify the definition of an Ada compilation system, definitions of an executable program and runtime library that fit within the conventions of the ARTEWG Framework are required. An executable program consists of directly translated source code plus the RTE. The ARTEWG Framework also provides the following definition for a runtime library:

> Runtime Library (RTL) - Set of all the predefined routines in a machine-executable representation that support all the functionality of an application program language that is not supported in code generated from the application programs."

Figure 1 illustrates a generalized Ada compilation system. At the top of the figure the compiler translates the Ada source code into three components: object code implementing the program design logic, object code implementing common coding conventions (consisting of data structures and control structure code sequences), and unresolved subroutine references. The heavy vertical bar separates the translated Ada program from the RTE components.

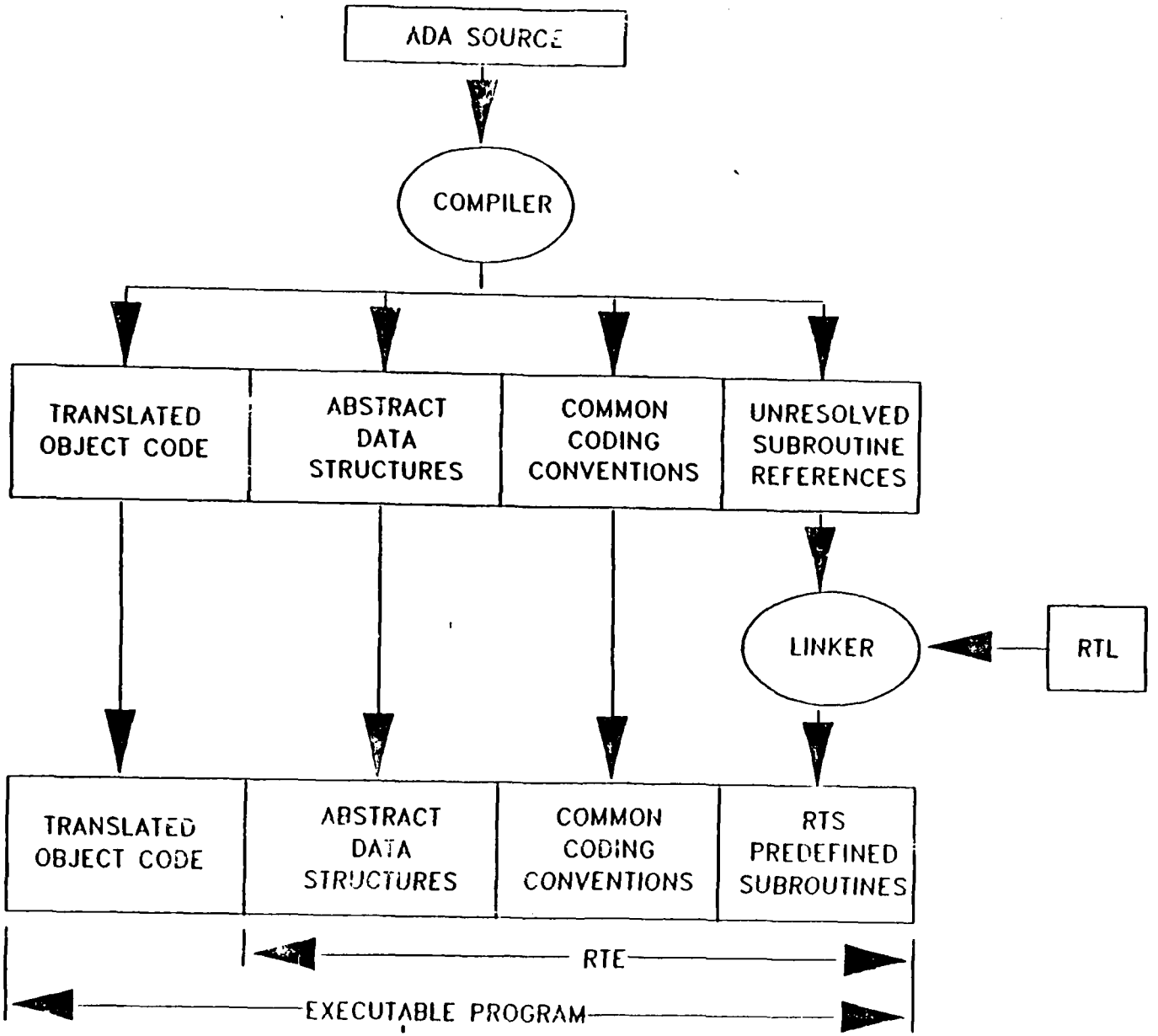


Figure 1 How an Ada Compilation System Produces an Executable Program

The linker resolves subroutine references from the RTL. The set of predefined subroutines extracted from the RTL by the linker make up the Runtime System (RTS). The ARTEWG definition of the RTS is:

> Runtime System - Set of predefined routines in a machine- executable representation that is selected by the Ada compilation system from a runtime library to support functionality of the application program not supported in the generated program."

At the bottom of Figure 1 the object representation of the translated Ada program and the RTE, now composed of the abstract data structures and common coding conventions and the RTS, make up the executable program.

Suppliers of Ada compilation systems may elect to decompose the same Ada construct in different ways. Depending on the limitations and capabilities of the target system, a compiler designer develops a model to guide the compiler development. The ARTEWG Framework (25) refers to this model as the *runtime execution model*. Implementation choices involve all components of the compilation process, including translated object code, abstract data structures, common coding conventions, and RTL subroutines.

As we will see in Section 3, the design of the runtime execution model may have important effects on the tailorability of the RTE. If RTE tailorability is to be possible, the user must be provided with a discipline, information, and tools to interact with the compilation process.

2.3 OVERVIEW OF TAILORABILITY

2.3.1 Candidates for Tailoring

To be able to tailor an RTE, one must first understand what portions of the compilation system are candidates for tailoring, as opposed to configuring. Figure 2 illustrates most of the possibilities. It should be

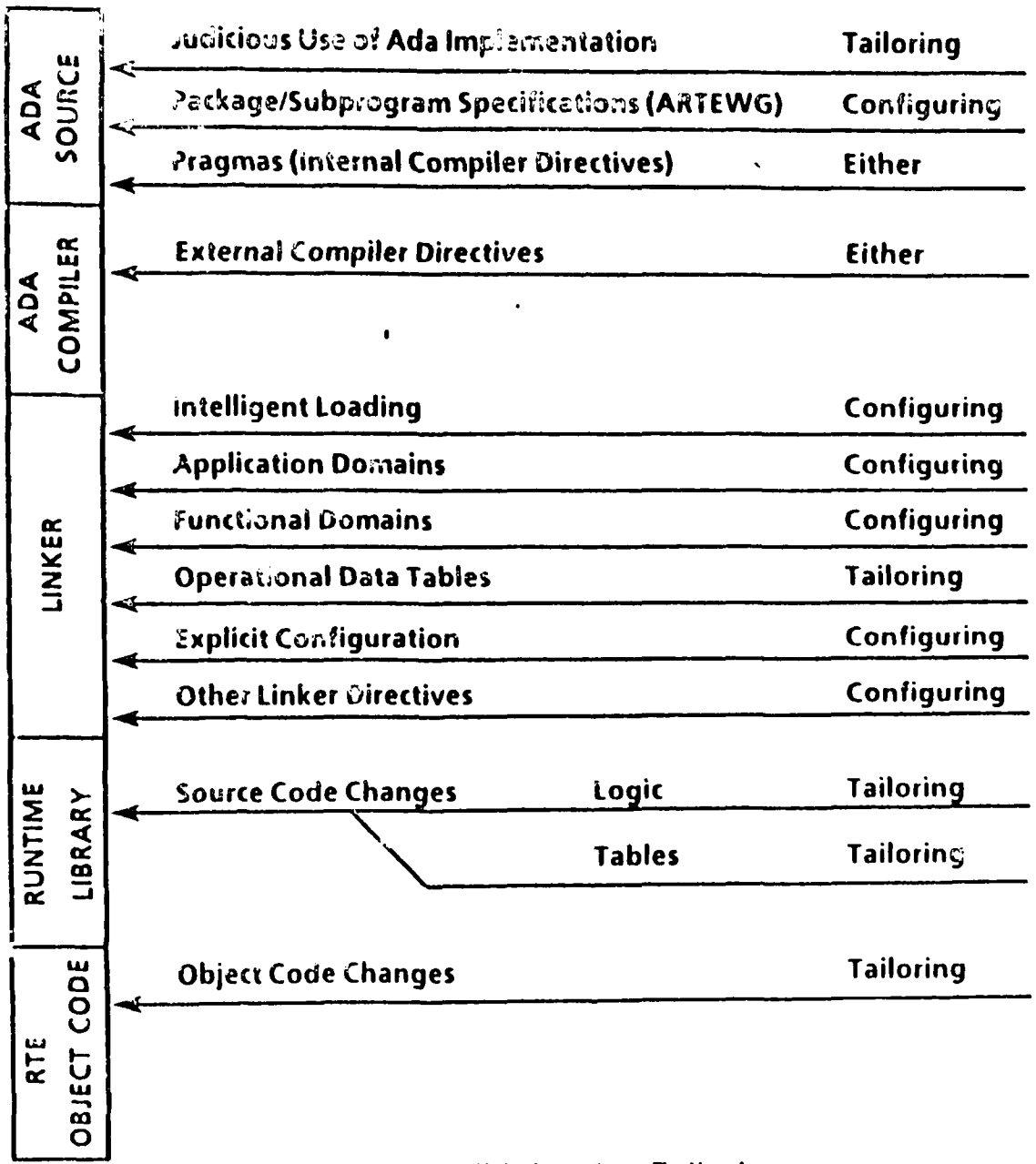


Figure 2 Candidates for Tailoring

remembered that the RTE consists of object code implementing common data structures, common coding conventions, and object code from the RTL.

At the top of Figure 2, it is shown that the Ada source code might be used to tailor the RTE. This would involve a situation where the first generation of an RTE proves to be inadequate in performance or some other measure, and the Ada source is modified to overcome this inadequacy. This is not an obvious way to tailor the RTE because it would require an intimate knowledge of how the compiler system would translate source code. However, it cannot be ruled out because it might be the first approach a knowledgeable programmer might try if assigned to tailor the RTE.

The use of common package specifications, like those proposed by ARTEWG (34) to request services of the RTE, must be considered configuring because their inclusion in the Ada source would result in pre-defined code units being added to the RTE.

The use of pragmas and other compiler directives may be considered to be tailoring, but only if their use produces changes in the common data or coding structures of the RTE; otherwise, it is configuring. But, again, this would require an intimate knowledge of the compilation system.

Most interaction with the linker would be configuring because configuring is the fundamental purpose of a linker. For example, intelligent loading is automatic configuring. It has also been suggested that specifying an application domain (e.g., smart weapons, aircraft guidance systems) or a functional domain (namely, sequential or concurrent) could signal the linker to selectively include only the relevant parts of the RTL. This would also be configuring. Figure 2 does, however, illustrate a special, important case of RTL tailoring, i.e., changes to tabular data. This data would be tailorable if it is embedded in the RTL source code.

Depicted at the bottom of Figure 2 are direct changes to the RTE object code. This can be considered tailoring, but it is not a very practical

approach and would likely be done only in a temporary situation such as for a quick patch, a debugging procedure or a temporary change.

Also shown in Figure 2 are changes to the Runtime Library or RTL. The RTS is part of the RTE and is built from the RTL, the RTL subprograms afford the most practical approach for developing common practices for tailoring the RTE. Only through the RTL is it possible for the user to work with source code and avoid getting into the intricacies of the compiler and the way it distributes functionality. That would be something neither the user nor the vendor would desire.

Providing tailorability features through the RTL allows a clean separation of responsibility that compiler vendors might find easy to support. However, much thought will have to be given to the RTL and how it can support tailoring through functionality, design, documentation, and tools. The remainder of this paper will assume that tailoring the RTE will be accomplished through source changes in the RTL.

2.3.2 Methods of Tailoring

Given that the RTL is the most logical focus for tailoring, what are the possible methods for tailoring it? Figure 3 illustrates three possible ways to tailor a hypothetical RTE based on the idea of a modular RTL: module change, module replacement, and module extension. Each of these ways exhibits various degrees of user and vendor responsibility.

Module 2.1 shows tailoring by direct changes to the body of the code. To accomplish such changes in-house, a deep knowledge of the implementation details of the module would be required and, to make such knowledge available, the vendor would have to provide comprehensive information. Such information would need to include detailed design documentation, training, and direct
assistance. While internal changes to code are not ideal,
modularity will minimize the tailoring required and the risk.

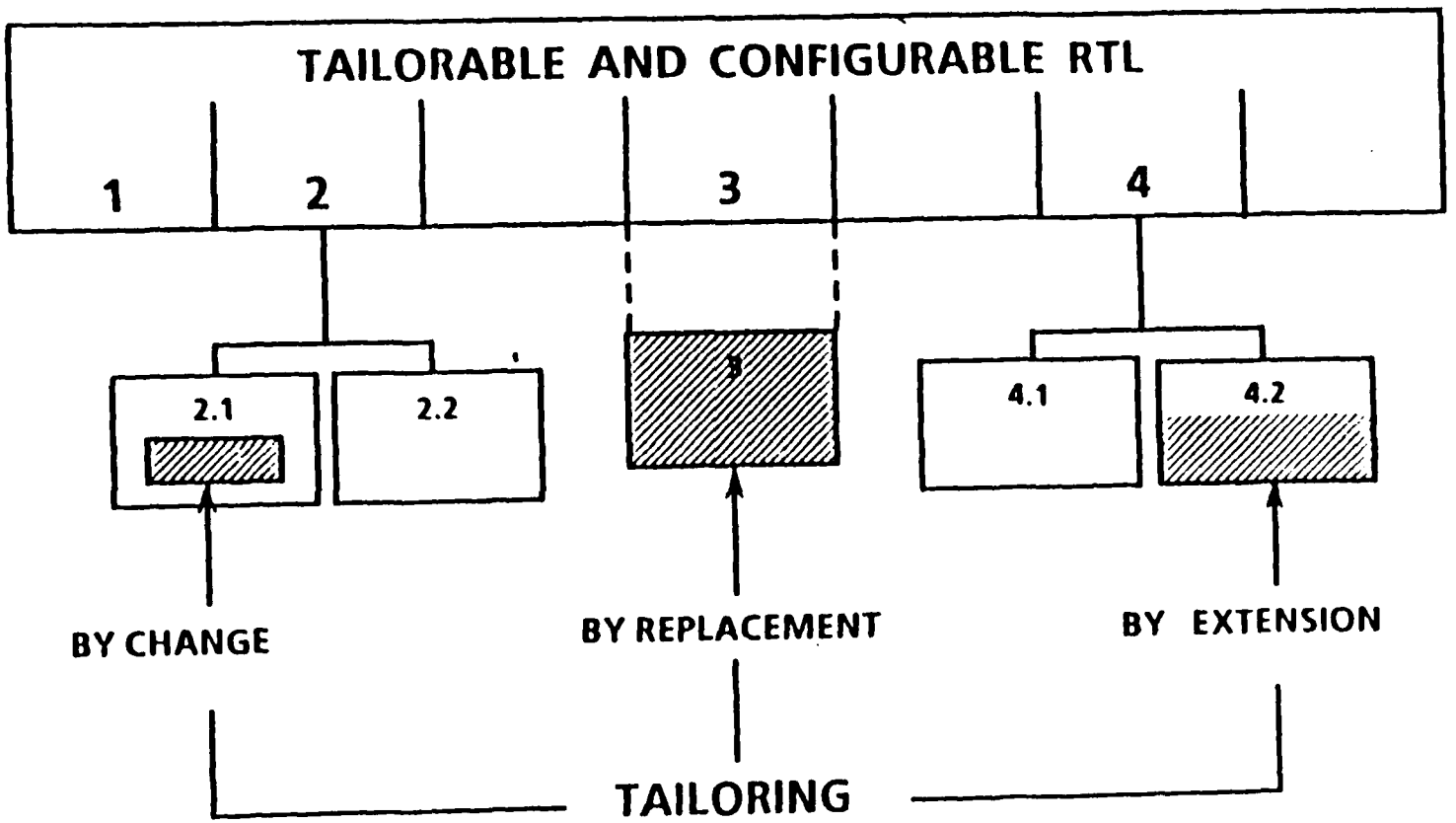


Figure 3. Ways to Tailor an RTL

At the other extreme, Module 3 is being tailored by completely replacing the vendor-supplied version with a new in-house version. This approach minimizes vendor involvement by requiring only that the vendor provide details of the specification part and an abstraction of the function. Ideally, the Ada program would not see any differences in the new version. In reality the vendor may need to supply caveats about side effects. This approach would not be practical if the RTE design did not exhibit sufficiently fine grained modularity.

In Module 4.2, tailoring is accomplished by extending functionality of primitive modules provided by the vendor. The vendor-supplied modules would provide some basic functionality and slots to add expanded functionality. The vendor may supply such slots through specifications and stubs or switches in the code. This approach would require that the vendor provide at a minimum high level design documentation and an expandable design that anticipates a wide range of special applications. It would avoid the necessity of a deep understanding of the higher level code provided by the vendor and provide a convenient way for the customer to add special functionality. This approach best supports the inclusion of customer supplied modules into the RTE.

3.0 HOW CAN ONE DETERMINE WHETHER AN RTE CAN BE TAILORED

The development of Ada compilers for bare machines is a relatively new science. There are many vendors producing such compilers, and their products exhibit a wide range of maturity and features. The parallel and relatively rapid evolution of microprocessors and other potential target machines has caused additional variations in available Ada compilers. Finally, the importance of the RTE to high performance embedded systems is now getting the attention it deserves, and vendors are considering features that provide user access to the RTE. With this in mind, this section examines criteria that might be used to evaluate RTE tailorability.

Criteria used to assess the tailorability of an RTE may be developed along two lines. One line would assist in recognizing design features of the RTE that promote tailorability. The other line would assist in recognizing services provided by compiler vendors to promote tailorability. These two distinct families of assessment criteria are discussed in the following subsections.

3.1 CRITERIA INDICATING DESIGN FEATURES WHICH PROMOTE RTE TAILORABILITY

The following criteria indicate that the compiler vendor has designed the compiler system in a way that simplifies tailoring. In contrast with the service criteria discussed below, whose presence or lack is fairly obvious, assessment of RTE design features that promote tailorability are likely to be more difficult to identify.

The RTE is a software system, and the application of criteria to determine design features that support tailorability can be compared to the measurement of software qualities such as correctness, maintainability, reliability, etc. There has been a good deal of research in this area that can be drawn upon to develop tailorability c eria. See, for example, Presson, et al. (38).

Arthur (11) gives a list of 11 software qualities and 22 criteria that are indicators of these qualities. Because the RTE is a software system, tailorability can be equated to the recognized software quality, flexibility. Arthur defines flexibility as, "How much effort does it take to enhance the program? (Can you change it?)." According to Arthur, criteria for assessing design features that support flexibility are clarity, concision, consistency, expandability, generality, modularity, self-documentation, and simplicity.

While there are existing measurement schemas for these criteria, they usually are intended for in-house quality control on large system developments and involve long questionnaires or complex formulas applied over the system life cycle. See, for example, the software quality assessment studies prepared for the Rome Air Development Center RADC (16) (31). These schemas are inappropriate for assessing the RTE tailorability of a candidate Ada compilation system because of time and cost considerations. What is needed are relatively high-level indicators that provide a quick assessment of the above design criteria. Considerations for applying these criteria are discussed in the following subsections:

3.1.1 Clarity

Clarity is a measure of the ease with which the system design can be understood. It is system clarity rather than code clarity that should be examined. The following considerations are relevant to assessing system clarity:

- o whether the RTE is well partitioned,
- o whether the RTE is hierarchically organized,
- o the complexity of RTE modules.

(1) A recommended approach to assessing the system clarity of the RTE is to attempt to discern the underlying organization of the RTE from the information described in Section 3.2.5 or other sources. If the RTE is well

partitioned, tailoring can be more easily localized, and the integrity of the RTE functionality will be more likely to be maintained.

(2) This organization may be broken down into different levels and may also provide common RTE configurations for specific processors and applications. A step-down approach from high-level functions to implementation/application specific routines can favor tailoring while still allowing for flexibility.

For example, the RTS may be divided into kernel routines and library routines. The kernel routines, which mask the machine dependencies and manage the machine resources, are also called executive routines. These are the routines that interact with the hardware and require processor dependent optimization. Library routines are responsible for the minimal runtime support required by the Ada Language Reference Manual. Any of these routines may be tailored by change, replacement, or extension, depending on application requirements.

(3) Other considerations are the number of RTE modules, the number of procedures and parameters in their interfaces, the number of variables shared by multiple modules, and the distribution of code among modules. If such an organization is relatively easy to recognize and seems to make logical sense, chances are that the RTE will be clear.

3.1.2 Concision

Concision is the ratio of the functionality provided by the system to the size of its source code. It is an attribute that indicates the tightness of the code. The following considerations are relevant to assessing concision:

- the size of the RTL for a typical application,
- the language in which the RTL is written.

(1) A rough estimate of the functionality of the system can be obtained by examining the size of the source code of a typical application it will service. This analysis should be tempered by the richness of functionality that is provided by the RTL.

(2) How much of the RTL is written in a high-level source language will affect the size of the source code. If the RTL is written in a high-level language, it will be more concise than one written in assembler.

3.1.3 Consistency

Consistency is a measure of the uniformity of the system design, its implementation techniques, and its documentation. The presence of consistency should be obvious from a spot examination of RTL source and documentation.

3.1.4 Generality

Generality is a measure of the degree to which system modules perform a wide variety of functions. Because of their role as part of a virtual machine, RTE modules should be general. The following considerations are relevant to assessing RTE generality:

- the presence of stubs for tailoring by extension.

(1) Generality can best be enhanced by use of stubs as suggested in the discussion of tailoring by extension. A good example of such generality is a data acquisition module that anticipates communication with application unique devices.

3.1.5 Modularity

Modularity is a measure of the degree to which separate functionality has been isolated into separate units of code. This criterion is essential

to both tailorability and configurability. The following considerations are relevant to assessing modularity:

- o the number of modules in the system,
- o the interdependencies between separate modules.

(1) Conceptually, an Ada RTE can be viewed as consisting of executive routines, storage management, tasking support, input/output, exception management, runtime library routines, and the mandatory Ada library units. Each of the above functions consists of packages and subprograms that can be broken down into modules with independent capabilities.

Forcing the runtime components into their own packages will further increase the integrity of modules. Decomposing modules into an increasing number of packages until the finest grained, practical level of modularity is achieved can facilitate the extension and replacement of units. Sufficient modularity can serve to minimize development costs associated with tailoring. Section 6.1 discusses these costs.

(2) Modularity is inversely related to coupling. Coupling, which can not be entirely eliminated, is a measure of the strength of interconnection between modules. The amount of coupling is affected by the level of modularity achieved and the functional distribution on which it is based. Generally, the choice will depend on performance/overhead requirements.

Loose coupling is essential if modules are to be tailored independently. By increasing the level of module granularity the risk of rippling and side effects due to tailoring can be minimized and even isolated. A finer grained modularity will mean loose coupling and greater overhead but make tailoring easier. A coarser grained modularity can provide an efficient tightly coupled RTE but lead to side effects when tailored. Coupling will also affect the compilation order. Hopefully, coupling between modules is documented and not extensive.

3.1.6 Expandability

Expandability is a measure of the ease with which additional functionality can be added to the system. The following considerations are relevant to assessing RTE expandability:

- o the presence of stubs for tailoring by extension,
- o documentation of techniques and hazards of expansion.

(1) The diversity of embedded applications will require runtime environments to be tailored such that functionality beyond that defined in the Ada Language Reference Manual be provided. Such functionalities can be provided through runtime support modules specifically tailored to the requirements. Some of the routines will need direct access to the hardware through the executive routines and should allow an implementation that is transparent to the application code. These extension modules may include device drivers, schedulers, performance monitors, and statistical/math functions. Section 2.3.2 discusses change, replacement, and extension as ways to tailor RTE modules.

(2) The specification of tailorable modules in the RTE should be sufficiently detailed to support one of these methods. There should be a detailed design for each module and warnings about side effects.

3.1.7 Self-Documentation

Self-documentation is a measure of the readibility of the system's source code. The following considerations are relevant to assessing self-documentation:

- o the presence of comments,
- o well chosen identifiers.

(1) The RTL source should be examined for sufficiently rich in-line documentation and other presentation features that make it more readable.

Experience has shown that extensive comments in particular contribute to understanding the design of a system.

(2) Design features of the code that enhance readability, such as the choice of type, object, and program unit names, should also be apparent.

3.1.8 Simplicity

Simplicity is the ratio of the size of a module to the number of variables, branches, loops, and program units within the module. It applies to a module that has been implemented in the most understandable manner. The following considerations are relevant to assessing an RTE's simplicity:

- o the size of a module,
- o the complexity of the module.

(1) The size of a module can be counted in many ways: one may consider total lines of code, total lines excluding comments, total semicolons, total semicolons excluding parameter separators, and so forth. Because the simplicity of a module depends on a ratio, the exact method of counting executable statements does not matter. What does matter is that this be done in a consistent manner.

(2) Things to consider in assessing the complexity of a module are the number of labels, loops, branches, nesting levels, and variables. Once again, exactly what one decides to count is not as important as that the count be done consistently across modules.

3.2 CRITERIA INDICATING SERVICES THAT PROMOTE RTE TAILORABILITY

The criteria described in this subsection indicate the availability of support for the user in becoming knowledgeable about the tailorable features of a particular RTE for accomplishing in-house tailoring. These criteria indicate that the vendor has not only recognized that tailoring might be needed but also is ready to assist user in tailoring. With the exception of

providing the source for the RTL (without which the RTL cannot be tailored at all), if the vendor does not provide this assistance, the presence of the above design features will be the only consideration in determining the tailorability of the RTE.

3.2.1 Availability of Source for the RTL

As discussed in Section 2.3.1, the best approach to in-house tailoring of the Ada RTE is through manipulation of the source code of the RTL. A compiler vendor should provide a means to obtain the source. Source for the RTL may be sold as a separate product by a vendor. The availability and cost of the RTL source should be considered in selecting a compiler even if, initially, there appears to be no need to tailor the RTE.

3.2.2 Availability of Vendor Support

The vendor should clearly provide a support staff for providing RTE tailoring services for specific customer applications. If a customer chooses to do his own tailoring, non-participatory support such as a "hotline" should be available. The relative costs for such services should be considered.

3.2.3 RTL Source Language

The source language of the RTL should be considered. It would be a definite asset if it were at least partially implemented in a high-level language, preferably Ada. If it is written in an assembly language, it should be an assembler that can be obtained from the vendor or a third party.

3.2.4 Sufficient Functionality of the RTL

Vendors may divide RTE functionality into a non-tailorable kernel and a tailorable RTL. The idea is that those functions assigned to the kernel are fundamental to computing on the target machine. If this is the case, it

must be assured that the kernel is sufficiently small and efficient and that the functions assigned to it will never need to be tailored. Conversely, functions that may require tailoring should be placed in the RTL and be available as source. The taxonomy of RTE elements described in Appendix A is one basis for determining the functionality provided in the RTL.

3.2.5 Documentation of Features that Support Tailorability

Vendor documentation should fully support any tailorability features that are provided. The design features of the RTE and their significance should be discussed. The components of the RTE and their relationships should be presented. The format of the RTL should be documented. Procedures and tools for editing should be described. Simple examples of tailoring the RTE should be provided. A User's Guide for tailoring would be a valuable feature.

3.2.6 Training

Availability of vendor training would be a important factor in choosing a compiler. The curriculum should address the topics covered in the documentation described in Section 3.2.5, experiences of the vendor with the RTE, examples and hands-on exercises. Instructors should be familiar with the vendor's RTE and RTL. The cost of vendor training should be considered.

3.2.7 Update Services

It should be recognized that Ada compilers for bare machines will improve fairly rapidly and that RTEs and RTLs will likewise become more efficient, flexible, and comprehensive in functionality. A positive feature of compiler products would be the availability of an update service. This update service would provide timely upgrades and documentation of the RTE and RTL that either lessen the need to tailor the RTE or serve to simplify tailoring. Cost is again a factor.

3.3 TAILORABILITY CHECKLIST

The criteria above and others that might be added can serve as the basis of a checklist that can be used to evaluate the tailorability of an RTE. An example is illustrated in Figure 4.

The checklist in Figure 4 has two sections, cost items and evaluation factors. The cost items consist of the cost of the basic compiler package and items that may be optional. For example, obtaining the RTL source may be a cost option. Costs must be considered to assure that they are not excessive. In the simple scheme presented, a compiler vendor is credited with a point if the item is available; and if there is a charge for that item, the cost is added to the cost of the basic compiler.

The evaluation factors serve to credit a vendor for additional features. For example, if the vendor's RTL is written in a high-level language, another point is credited. After the checklist is completed, the credited points are added. This gives a measure of an RTE's tailorability. To provide a cost/benefit measure the number of points is divided into the total cost. The cost/benefit ratio has value only when dealing with compilers with approximately the same features. If the features are needed and only one vendor provides them to the extent that is needed, there is no choice.

In designing such a checklist there is likely to be debate over what is included and whether each item's relative worth should be considered (i.e., whether an item's value should be weighted). It is also recognized that some items, like whether RTL source is available, are readily answered. Other items, like modularity, will require some review of documentation and perhaps a subjective judgement. However, such a checklist should be kept simple or it would be too cumbersome and time consuming to be useful. Even highly subjective questions should, at least, stimulate a question to a vendor or a look at the vendor's literature.

COST ITEMS	AVAILABILITY	COST
Ada Compiler	[✓]	$80,250
RTL Source	[✓]	20,000
RTL Compiler/Assembler	[✓]	NC
Documentation	[✓]	NC
Update Service	[ ]	—
Vendor Support	[✓]	32,500
Training	[✓]	10,000
TOTAL COST		$142,750

EVALUATION FACTORS	AVAILABILITY
RTL Language	
High Level Language	[ ]
•	[ ]
•	[ ]
Documentation Coverage	
RTE Tailorability Features	[ ]
RTE Configuration	[✓]
RTL Format	[✓]
Editing, Compiling and Linking	[✓]
Examples	[ ]
Separate Tailoring Guide	[ ]
•	[ ]
•	[ ]
Training Curriculum	
RTE Tailorability Features	[ ]
RTE Configuration	[✓]
RTL Format	[✓]
Editing, Compiling and Linking	[✓]
Examples Provided	[✓]
Practical Experiences Provided	[✓]
Hands-on Exercises	[✓]
Experienced Instructors	[✓]
•	[ ]
•	[ ]
Update Services	
Documentation	[ ]
Source Code Patches	[ ]
Extensions	[ ]
•	[ ]
•	[ ]

Figure 4 Tailorability Checklist

EVALUATION FACTORS	AVAILABILITY
RTL Functionality	
Memory Management	[✓]
I/O Management	[✓]
Exception Management	[✓]
Task Management	[✓]
Interrupt Management	[ ]
•	[ ]
•	[ ]
Design Features	
Clarity	[✓]
Concision	[ ]
Consistency	[✓]
Expandability	[ ]
Generality	[✓]
Modularity	[✓]
Self Documentation	[✓]
Simplicity	[ ]
•	[ ]
•	[ ]
TOTAL POINTS	[25]
COST / BENEFIT (TOTAL COST / TOTAL POINTS)	142,750 / 25 = 5710

Figure 4 Tailorability Checklist (Cont.)

4.0 WHAT CAN ONE DO TO TAILOR AN RTE

The following subsections describe how to tailor an Ada RTE. First, an analysis of opportunities for tailoring is presented. Next, general steps for tailoring are provided. Finally, a set of examples of tailoring is presented. For the taxonomy of the RTE on which these subsections are based, see Appendix A.

4.1 OPPORTUNITIES FOR TAILORING WITHIN THE RTE

There are a number of areas in an RTE that are candidates for tailoring. In this section, those areas are delineated, and the various tradeoffs that must be considered are discussed. No evaluation of the relative difficulty of performing each type of tailoring or of a particular compiler vendor's implementation is provided here, although what must be accomplished before tailoring can begin (e.g., obtaining the source of the RTL) is discussed where appropriate.

4.1.1 Task Activation

(1) Task creation may be static, or it may be dynamic. In situations where it is known-at compile time that there are going to be a limited number of static tasks, it may be preferable to tailor the system by pre-elaborating those tasks and creating the data structures within the RTL kernel for activation of those tasks immediately upon program execution, rather than requiring that the tasks wait until program start-up to be created.

The same effect could be achieved by declaring tasks within library packages, but sometimes it may not be possible to do this without violating the application design. In this case, tailoring the RTE might be preferable.

4.1.2 Task Scheduling

Real-time embedded systems often have timing constraints that require a particular operation or set of operations to be completed by a certain time in order for the software to meet its requirements. The ability of the software to adhere to those constraints can be significantly affected by processor scheduling. Although reliance upon the rendezvous mechanism to coordinate execution of various tasks is preferred to the use of relative priorities to accomplish this objective, a number of opportunities remain within the choice of a scheduling algorithm and the characteristics of a particular algorithm of processor allocation to improve performance for an embedded application.

(1) One of the most common scheduling algorithms is the time slice, or round robin, algorithm. In this situation, a task gets a particular slice of the processor in which to run, and is suspended during other portions of the processing period while other tasks may be running. It is possible to alter the amount of time covered by each slice of processing time, which may improve performance by reducing the number of context switches that occur during a particular application. Time slice scheduling requires that the CPU be interrupted. It must be carefully controlled to prevent the interruption of critical sections of code. Time slicing also requires access to the system clock.

Sometimes the amount of time covered by each slice can be altered by a runtime parameter, but this may not always be the case. Some systems, for example, may not provide this capability. In other cases the amount of time needed per slice may be smaller than the minimum provided as a runtime parameter. In either case, tailoring the RTE's time slicing mechanism would be necessary.

The problem with the time slicing algorithm is that, taken strictly, it is incompatible with having task priorities. Ada requires that if two tasks with different priorities are ready for execution at the same time, the

higher priority task must be executed first (LRM 9.8 (4)). A round robin approach is only possible when all tasks are of the same priority.

(2) Another popular scheduling algorithm is the rate monotonic algorithm. This algorithm makes its scheduling decisions on the basis of job importance, periodicity, and average and worst case execution times. One consequence of this approach is that if a high priority task has not completed its execution within a certain time period, it may be preempted in favor of a lower priority one. For this reason, the rate monotonic algorithm is strictly incompatible with the rules for handling priorities in Ada (LRM 9.8 (4)).

One way to overcome this limitation of the rate monotonic algorithm, suggested by Cornhill (21), is to assign the same priority to all tasks. Because there are no restrictions on the order of execution of tasks with the same priority (LRM 9.8 (5)), tasks can be scheduled in any order. What would otherwise be priority information can be passed to the scheduler by some other mechanism, such as another pragma.

(3) At the opposite extreme from the time slicing approach is the first-in-first-out mechanism. This would allow a task to run to completion before another task might have the opportunity to be run. In applications where no asynchronous events will occur, this purely cyclic scheduling algorithm may be the most efficient. Because, however, real-time systems do contain asynchronous events, the first-in-first-out algorithm is unsuitable for real-time systems.

4.1.3 Interrupts

(1) Section 13.5.1 of the LRM states that interrupts are treated as entry calls by a hardware "task" whose priority is higher than that of the main program and any user-defined task. Implementing a hardware interrupt involves mapping a hardware signal to a high-priority Ada entry call that looks like an ordinary entry call (timed, conditional, or normal) to the

runtime system. The following semantics for this type of entry call may be tailored for a particular application:

- o parameters,
- o storage area and exception raising,
- o scheduling actions (which may not be needed/invoked),
- o enabling/disabling of interrupts,
- o priorities of nested interrupts,
- o where the rendezvous is executed (e.g., on a stack),
- o restrictions on the terminate alternative,
- o actions taken when an interrupt is not serviced immediately.

Interrupt handling via interrupt entries is appealing because it provides flexibility in the handling of each interrupt, and may give the application program a measure of control over the outcome of the interrupt. All the above mentioned semantics may be tailored for each unique interrupt by employing tasking constructs.

Performance may be sacrificed when employing this scheme. The interrupt latency (the time it takes the RTE to respond to an interrupt), and the overhead (the time it takes for the interrupt to be handled) are high because of the dependence on the tasking scheme. There may also exist implementation restrictions on interrupt entries that may limit some of the desired customizations. The scheduling behavior will also be a factor.

Alternatives to using Ada interrupt entries include:

- o coding the entire interrupt handler in Ada (e.g., subprogram, task) and specifying its starting memory location through an address clause,
- o implementing the interrupt handling code in another language and linking it with the Ada code via the pragma INTERFACE,
- o implementing the traditional interrupt service routines through the use of interrupt vectors,
- o implementing interrupts using "fast" interrupt techniques.

(2) The use of an address clause will reduce the overhead and may increase the performance but will limit the flexibility and transportability of the RTE. In some cases, the user may not be able to control the outcome of an interrupt (e.g., enable the interrupt, delay it, handle it specifically).

(3) Implementing the code in another language will increase the performance if the code is written in assembler to provide maximum optimization. This utility will be very machine dependent and must be bound differently for each machine.
Flexibility and control are sacrificed.

(4) The use of interrupt vectors yields the best performance and incurs the least amount of runtime overhead; however, there is usually no control over the outcome of an interrupt by the user. This technique may also require the direct management of the machine's low level hardware.

(5) The implementation of "fast" interrupts (rendezvous that are treated like procedure calls and executed on the caller's runtime stack) can eliminate tasking overheads and reduce interrupt response time; however, this may require that all runtime stacks may be visible to the entire system, which may not be possible due to memory limitations.

The choice of customization and implementation mechanism will likely be influenced by the required interrupt response time. In a system where the rendezvous time is slow, or the timing for the delay statement and for timed entry calls is inaccurate, interrupt tasks may not be appropriate. Special interrupt service routines may be implemented in situations where an interrupt task implementation causes a substantial increase in the runtime overhead or where certain hardware interrupts must be handled immediately. It is likely that a combination of interrupt handling schemes will be the best customization. For example, interrupts related to exceptions or I/O can be serviced by interrupt tasks to be passed to the appropriate RTL

modules for processing. Interrupts that can be handled transparently by the RTE or have timing requirements can be serviced by special service routines.

4.1.4 Storage Allocation

One of the greatest limitations in embedded systems is the amount of memory that is available for executing software (some of the other major limitations being processor speed and backing store devices). This limitation not only constrains the size of the program being developed (in terms of the amount of space available for code), but also constrains the amount of space available for data. (Addition of even a minimal amount of code and data for the RTS can further constrain the RTE.) As a result of the memory limitations, most RTEs do not include mechanisms for virtual memory, and so are limited to the available physical memory.

(1) Two of the most common features that are available for tailoring are the size of the runtime stack and the size of the heap. The stack is generally used during procedure and function calls to store parameters upon invocation, and as a temporary data location to store results upon return. The heap is the area of memory in which temporary variables (such as are created during expression evaluation) and other allocated variables (using the NEW construct) are stored. In a general layout, the code and static data are stored at one end of memory, and the heap and stack are then placed at opposite ends of the remaining memory and allowed to grow toward one another. By being able to raise the top of the stack, it is possible to allow programs that might have deeper calling structures to execute, while enlarging the size of the heap might allow programs that need to maintain greater amounts of data to run.

Many compilers currently allow the user to configure the size of the stack and the heap through user-defined addresses. However, if the compiler does not provide this capability, tailoring these values would be necessary.

(2) In addition to the size of the stack and the heap, a significant number of opportunities for tailoring exist with respect to memory

management within the heap. At the grossest level, rather than relying upon the memory management of the RTE, the program could manage memory itself for any of the allocated types by not deallocating the memory and maintaining its own internal free list. This might have the advantage of providing faster allocation times during the execution of the program for long-lived programs, provided that sufficient memory is available to set it aside when not in use. On the other hand, a bad memory management algorithm can severely degrade system performance, so this area of tailoring is somewhat risky.

(3) If it is preferable for the system to manage memory, there remain additional characteristics of such a management scheme that can affect the execution performance of a particular application. One such issue is the block size of memory allocation. Rather than allocate memory blocks that are precisely the size required for a particular data item, it may be preferable to allocate them in standard chunks, such as 512 bytes, in order to avoid excessive fragmentation of memory. In situations where fragmentation does occur, it must then be determined when to perform compaction, either upon memory exhaustion, upon each de-allocation, or at some time dynamically specifiable by an RTE call from the program in execution.

(4) The presence of tasking in such an environment further complicates the matter of storage management by the need for separate stacks, and potentially multiple heaps for each task in execution. This storage space must also be considered for reclamation upon termination or completion of the task, as any other storage.

(5) One other area in which storage management is a consideration, not necessarily a function of the RTE, is that of generics. If the compiler is capable of performing sharing of the code segments for multiple instantiations of a generic, then the amount of code that must be included in a runtime executable may be significantly smaller than if such sharing had not been performed. A storage management mechanism that does not

provide code sharing could be tailored to do so, but this would be a significant undertaking.

4.1.5 Exceptions

(1) In accordance with the Ada standard, the RTE detects exceptional situations and raises the corresponding predefined exception during program execution. Suppression of these runtime checks can be accomplished via the pragma SUPPRESS, if it is supported, in order to reduce the execution overhead of performing them and the associated code size overhead. However, suppressing all checks of a given kind may be too extreme for some situations. For example, the user may wish to suppress constraint checking on all variables except one. In cases such as this, it may be necessary to tailor pragma SUPPRESS to provide some level of error detection. The tradeoffs that must be considered when deciding to suppress the runtime checks include whether or not the detection mechanism can be disabled, whether the execution time and the amount of object code will in fact decrease, the effects of the loss of runtime error detection, and the amount of additional application code that may be needed to replace the suppressed checking.

Limited error checking can also be performed at the application level, but in some cases it may be easier to tailor pragma SUPPRESS rather than to explicitly check for errors throughout the system. Examples of this would be when variables that need checking occur frequently throughout the system, or when the system is likely to require modification or enhancement at a later date. In these cases it may be preferable to perform error checking in one place, rather than run the risk that it be overlooked accidentally.

(2) Tailoring the exception management mechanism may be complicated by the fact that many of the runtime checks are performed by compiler-generated code (e.g., constraint checking). However, because of the execution and code size overhead incurred in support of the exception mechanism, customizing it can result in substantial gains. Possible opportunities for tailoring within the exception handling mechanism include

- handling exceptions locally without propagation,
- propagating the exception to the frame that is at the head of the call chain and handling it there,
- handling the exception by an intermediate frame,
- propagating the exception back anonymously to be eventually trapped via a WHEN OTHERS clause,
- providing corresponding error handling code in-line,
- providing a special error processing package for inclusion when an error is likely,
- restricting the use of predefined exceptions.

(3) For certain applications it may be possible to remove the exception processing module from the runtime library, thus accomplishing what the pragma SUPPRESS accomplishes, without the need for implementing it. For example, as in the case of the NUMERIC_ERROR, runtime checks can be provided by hardware. On the other hand, if any portion or all of the RTE's exception processing module is removed and not replaced by hardware checks, the burden falls on the application program to perform its own runtime error checks. Applications may have to develop code with both pre- and post-subprogram checks for examining the validity of the program's execution state, or passing error return code parameters between both subprogram and entry calls so that their values can be checked upon return from the call. For this reason, removing the exception handling mechanism should be reserved only for extreme situations.

4.1.6 Input/Output

Because many embedded systems have only limited secondary storage requirements, it is not uncommon to find the Ada compilers for embedded systems lacking in Chapter 14 features. Not having routines for those features in the RTL reduces the size of the RTL; however, some small amount of I/O, particularly a limited amount of TEXT_IO or some INTEGER_IO or

FLOAT_IO may be found to be quite beneficial, specifically during development and testing.

(1) For the most part, however, embedded systems tend to be I/O intensive and often need low-level interfaces to hardware devices. Tailoring the I/O will require access to I/O ports, to control, status, and data registers, to direct memory access controllers, and to mechanisms enabling and disabling device interrupts. Low-level asynchronous I/O operations with hardware devices can be implemented using an interrupt driven scheme if the RTE is able to efficiently handle interrupts (e.g., the RTE supports task interrupt entries to allow bindings, via address clauses, between the task entry and the hardware device). The performance of an interrupt driven scheme will be affected by the interrupt latency, runtime overhead, and the speed of the interrupt scheme employed. Implementations may place restrictions on the use of an interrupt driven scheme. Specifically, they may not allow entries to have parameters, may restrict the types of entry calls allowed (e.g., normal, timed, conditional), or may have a scheduling scheme that degrades the performance of I/O.

(2) The implementation of package LOW_LEVEL_IO provides the interface with a hardware device through the SEND_CONTROL and RECEIVE_CONTROL procedures. These two procedures are designed for the purpose of being overloaded for various device types and data formats; the specifications for their two parameters, the device type and the format of data packets, depends upon the physical characteristics of the many different devices. Tailoring may specialize the device interface to allow the application program to completely control a specific device and thus exploit the hardware through the use of its instruction set.
As in tailoring any I/O package, tailoring the LOW_LEVEL_IO module is preferable to modifying the overall scheme of the RTE's I/O. This is because fine tuning the LOW_LEVEL_IO module for a particular device would not require changes to other modules. This technique is less complex and reduces the possibility of side effects, but it does limit transportability.

An RTE that does not implement I/O packages forces the application programmer to develop device specific interfaces either in Ada, using representation pragmas (e.g., PACK) and clauses (e.g., LENGTH), or in another language provided that the interfaces can be linked with Ada code via the pragma INTERFACE or code statements.

(3) When there is a need to add a new peripheral device to the RTE, a driver for the device must be incorporated to the runtime system. Writing a device driver task requires knowledge of the I/O tasking rules and the I/O Management subsystem. This device driver will likely need to be incorporated into one of the I/O packages that create device driver tasks during the RTE's elaboration. To do this will require tailoring of the RTE. The device driver will provide information such as an interrupt, an I/O port, and jump table entry (to provide an entrance into the I/O low-level/channel programs) that the RTE will associate with a particular device. The RTE enables and handles the device interrupt, sends control signals to and requests status from a device, and moves data to and from the data registers or the I/O memory. The channel programs provide the interface between the CPU and the peripheral device; all I/O will go through a channel program. These channel programs execute the required operations for the transferring of data. High level and text level packages may interface with the device drivers directly by employing such Ada facilities as address specifications and machine code insertions.

4.1.7 Target Specific Functions

(1) There are a number of target specific features that provide an opportunity for tailoring the RTE. The primary ones are the presence (or lack) of particular hardware co-processors (such as floating point processors) or the use of emulation libraries to simulate such co-processors. Such (or similar) libraries may be found to provide insufficient performance, and in situations where actual hardware co-processors are not readily available, a rewrite of portions of the existing library routines may be required in order to achieve the requisite performance. These situations are rare, however.

4.2 GENERAL STEPS INVOLVED IN TAILORING

Because every tailoring effort is unique in many respects, what one must do to tailor an RTE must be determined on a case-by-case basis. Nevertheless, in this section we will present the following general guidelines for tailoring any RTE (which, as we saw in Section 2.3.1, is really a matter of tailoring the RTL).

(1) Acquire the RTL source code. With the exception of pragmas and compiler directives, tailoring is impossible without the RTL source code (see Section 2.3.1). For more information on acquiring the RTL source code, see Section 3.2.1.

(2) Develop or acquire a full suite of tests for the RTE.

(3) Run the tests.

(4) Record the results.

(5) Save a copy of the original source code and all test programs. This is very important, as tailoring the RTE is an inherently risky business. If the first effort should fail, it is crucial that the original starting point be available for another try. See Section 5.2 for the philosophy behind this and the preceding three recommendations.

(6) Locate the code to be (a) changed, (b) replaced, and/or (c) extended.

(7) Identify all other areas of the source code on which the tailored area(s) depend(s) and which need to be changed as a result of the tailoring. Refer to Section 5.1 for a discussion of the kinds of thing to look for.

(8) Tailor these areas.

(9) For each tailored area, repeat steps (7) through (8) until all areas affected by the tailoring have been changed.

(10) Recompile as much of the RTL as necessary. It is usually not necessary to recompile the entire RTL. Link and load.

(11) Test the system with the new RTE.

(12) Compare the results with those recorded in step (4).

(13) If further tailoring is required, repeat steps (6) through (10).

For more detailed discussions of steps (6) through (10) as applied in specific cases, see the examples below.

4.3 EXAMPLES OF TAILORING

This section provides some examples of modifications that might be undertaken with a practical emphasis on tailoring an application under various compiler vendors' implementations. The particular areas that these examples address are the following:

- o adding a new device driver,
- o changing the runtime layout of available memory,
- o correlating the runtime system with the system clock,
- o selecting a processor scheduling algorithm.

Clearly these examples illustrate the entire range of tailoring activity (see Section 2.3.2). Adding a new device driver, for instance, is mostly an example of tailoring by extension. Although some change of the RTE is required, most of the work involves adding code to handle communication with the new device. The other three examples are much more complicated. They are almost purely examples of tailoring by change or by replacement, depending on the extensiveness of the alteration.

The first example illustrates steps (6) through (10) in detail. Step (9) is omitted because no iterations are needed in this case. The other three examples illustrate step (6) for a variety of other areas.

4.3.1 Device Driver

This section provides an example of what is involved in tailoring an RTE by describing the process of adding a new peripheral device, the EPSON FX-286 printer, to the Ada Language System/Navy's (ALS/N) runtime system for

the AN/UYK-43 (Ada/L) embedded computer. This example is based on information provided in the Ada/L Program Support Environment Handbook (5).

The necessary steps when adding a new peripheral device to the Ada/L environment are the following:

Step (6) Write a new package to the Physical I/O layer to support the new device type.

Step (7) Modify package IO_DEFS and subprogram FILE_IO.IO_REQUEST.

Step (8) Write a new device driver for the device.

Step (8) Write a channel/chain program for the device.

Step (10) Compile and Link the Executive as required.

Step (10) Compile and link the RTL as required: compile the package spec and body; assemble and import the channel/chain programs for the configured channels.

Step (10) Configure the new peripheral device using the Exporter, and Export the system.

The following packages must be visible to the device driver: SYSTEM contains the definition of configuration dependent characteristics and provides system dependent logical routines and conversion routines; IO_DEFS contains the implementation- dependent object and type declarations used in I/O Management; ADA_RTEXEC defines the components of the RTE for Ada/L that are visible to the Runtime Support Library through the RTEexec_Gateway or are executed in response to a target interrupt; RTEXEC_GATEWAY provides the means to invoke the services of the RTE; FILE_IO provides services for basic file operations including open, close, read, write, and reset.

Step (6) The Device Type Package

Because the EPSON FX-286 printer is a device of a class that is not supported by Ada/L, a new package to support this new device type must be written and added to the logical grouping (Physical I/O) of Ada I/O device packages. This package must have as a minimum the specification level

subprogram MAKE_REQUEST with the File_Information_Block pointer as a parameter.

ADA_RTEXEC.CONFIGURE_IO.CREATE_DEVICE_DRIVERS allocates one task (e.g., FX286_DD) for every device driver (e.g., FX286_DEVICE) that is implemented.

ADA_RTEXEC.CONFIGURE_IO.INITIALIZE_DEVICE_DRIVERS calls the SET_UP entry of each device driver task to perform device/channel initialization.

The PRINTER_IO package provides all the necessary data structures and support code to provide an Ada program with an interface to a printer driver as an I/O peripheral device using high-level Ada I/O packages DIRECT_IO, SEQUENTIAL_IO, and TEXT_IO.

The package specification is as follows:

```
WITH FILE_IO;

PACKAGE PRINTER_IO IS

  PROCEDURE MAKE_REQUEST
  -- determines the printer I/O function request and
  -- gives processing control to appropriate subprogram

     (stream : IN OUT file_io.stream_id_prv);
               -- identifies the File_Information_Block
               -- associated with this printer request

END PRINTER_IO;
```

The package body is as follows:

```
WITH SYSTEM, RTEXEC_GATEWAY, IO_DEFS, FILE_IO;
USE SYSTEM, IO_DEFS;

PACKAGE BODY PRINTER_IO IS

   printer_error : EXCEPTION;

   TYPE printer_operation_enu IS
      -- identifies the I/O operations provided by the printer

   (write  --  Printer operation
   );

   empty_buffer_length : integer := 1;
```

```
-- the length of the empty buffer sent to the printer by
-- TEXT_IO when a new line is requested.

TYPE null_buffer_rec IS
-- types and objects required for writing an empty buffer
-- defined as a record for use with 'ADDRESS call

   RECORD
      null_buffer : integer := 16#20202020#;
   END RECORD;

printer_empty_buffer : printer_io.null_buffer_rec;
                       -- object to perform 'ADDRESS on

PROCEDURE WRITE

-- WRITE will build an I/O Request_Block to output the given
-- amount of data to the printer

   (stream : IN OUT file_io.stream_id_prv
             -- identifies the File_Information_Block
             -- associated with this printer request
   ) IS

   iorb: io_defs.iorb_access RENAMES stream.iorb;
         -- the file's I/O_Request_Block

   BEGIN

      IF (iorb.data_length > 0) THEN
         -- data exists to transfer

         -- build the I/O Request_block
         iorb.function_request :=
            printer_io.printer_operations_enu'POS(write);

         -- send request to executive
         RTEXEC_GATEWAY.MAKE_REQUEST(iorb);

         IF (iorb.status /= io_defs.ior_ok) THEN
            RAISE printer_error;
         END IF;

      END IF;

      -- determine if new line character is needed
      IF (stream.new_line) OR ELSE
         (iorb.data_length = 0) THEN

         -- prepare I/O_Request_Block for a write operation

         -- load the new line buffer's address and length into
```

```
         -- the I/O_Control_Block
         iorb.data_location   := printer_empty_buffer'ADDRESS;
         iorb.data_length     := empty_buffer_length;

         iorb.function_request :=
            printer_io.printer_operations_enu'POS(write);

         -- send new line character
         RTEXEC.GATEWAY.MAKE_REQUEST(iorb);

         IF (iorb.status /= io_defs.ior_ok) THEN
            RAISE printer_error;
         END IF;

      END IF;

   END WRITE;


   PROCEDURE MAKE_REQUEST
   -- determines the pointer I/O operation requested from the
   -- requested_operation field of the File_Info_Block and give
   -- processing control to the appropriate subprogram in
   -- PRINTER_IO; it builds the command to be sent to
   -- CHANNEL_IO.

      (stream : IN OUT file_io.stream_id_prv
                -- identifies the File_Information_Block

      ) IS

   BEGIN

      CASE stream.ALL.requested_operation IS

         WHEN io_defs.write_request =>
            PRINTER_IO.WRITE(stream);

         WHEN OTHER =>
            stream.ALL.operation_result := io_defs.ior_ok;

      END CASE;

      -- return with an operation result of ok
      stream.ALL.operation_result := id_defs.ior_ok;

   EXCEPTION

      WHEN printer_error =>
         stream.ALL.operation_result := io_defs.ior_device_err;

      WHEN OTHERS
```

```
        -- raise the appropriate exception
        RAISE;

   END MAKE_REQUEST;

BEGIN  -- PRINTER_IO ELABORATION

   -- initialize the device drivers
   RTEXEC_GATEWAY.INITIALIZE_DEVICE_DRIVERS;

END PRINTER_IO;
```

Step (7) Modifying FILE_IO.IO_REQUEST

Subprogram IO_REQUEST within package FILE_IO must be modified to call the new package PRINTER_IO. Devices are identified by the enumeration type IO_DEFS.PERIPHERAL_DEVICE_ENU. This enumeration type must be modified to include the FX286_DEVICE (this trivial modification is omitted here). IO_REQUEST contains a case statement on this enumeration type that will call the appropriate Physical I/O package MAKE_REQUEST subprogram for the devices supported.

```
PROCEDURE IO_REQUEST
-- determines from the device type, the Physical_IO package
-- required to complete the I/O request, and calls the
-- appropriate subprogram to perform the requested function

   (request : IN io_defs.io_operations_enu;
              -- defines the I/O function requested

    stream : IN OUT file_io.stream_id_prv
             -- provides access to the File_Info_Block

   ) IS

   iorb : io_defs.iorb_access RENAMES stream.iorb;
          -- the I/O request block associated with this file

BEGIN

   -- set the request into the File_Info_Block for future use
   stream.requested_operation := request;

   -- set up fields of the I/O_Request_Block
   iorb.unit_number  := stream.unit_number;
   iorb.device_id    := stream.device_id;
```

```
    iorb.device_type  := stream.device_type;

    -- determine the type of device driver to invoke according to
    -- the value in the device_type field of the
    -- File_Information_Block
    CASE stream.device_type IS

       WHEN io_defs.rd358_device =>
          TAPE_IO.MAKE_REQUEST(stream);

       WHEN io_defs.uyh3_device =>
          DISK_IO.MAKE_REQUEST(stream);

       WHEN io_defs.usq69_device =>
          TERMINAL_IO.MAKE_REQUEST(stream);

       WHEN io_defs.fx286_device =>
          PRINTER_IO.MAKE_REQUEST(stream);

       WHEN OTHERS =>
          stream.operation_result := ior_device_err;

    END CASE;

END IO_REQUEST;
```

Step (8) The Device Driver

The device driver task contains seven entries (SET_UP, MAKE_REQUEST, EXTERNAL_INTERRUPT, EXTERNAL_FUNCTION, OUTPUT_MONITOR, INPUT_MONITOR, and IOC_CP_INTERRUPT). The last five entries are used as a means of informing the device driver task that an I/O operation has completed. These represent the five types of I/O monitor interrupts that can be generated by the AN/UYK-43 IOC. Any modification to the specification of this package will require corresponding modifications in ADA_RTEXEC.CONFIGURE_IO and compilation of the RTE.

The package specification is as follows:

```
PACKAGE FX286_DEVICE IS

   TASK TYPE FX286_DD IS

      ENTRY set_up
            su_ptr : IN ada.rtexec.configure_io.
```

```
                        set_up_blk_access);
                        -- the set_up_blk contains: the primary
                        -- channel, the secondary channel,
                        -- secondary status, max-transfer size,
                        -- and max-retry count

      ENTRY make_request
            iorb2_ptr : IN ada.rtexec.configure_io.iorb2_access);
                        -- iorb    : io_defs.io_request_block
                        -- channel : io_defs.channel_range_int
                        -- physical_data_location :
                        --              physical.address

      ENTRY external_interrupt;

      ENTRY external_function;

      ENTRY output_monitor;

      ENTRY input_monitor;

      ENTRY IOC_CP interrupt;

   END FX286_DD;

END FX286_DEVICE;
```

The package body is as follows:

```
WITH SYSTEM, IO_DEFS, ADA_RTEXEC, UNCHECKED_DEALLOCATION;

USE IO_DEFS, SYSTEM;

PACKAGE BODY FX286_DEVICE IS

   TYPE io_operation IS
      -- this enumeration type is to indicate which device
      -- type channel programs to execute

      (write,
       -- send EF, activate output buffer, request EI

       initialize);
       -- send enough EF to accomplish initialization

   TASK BODY FX286_DD IS

   pri_channel    : io_defs.channel_range_int;
   ei_word        : INTEGER;
   ei_accept_flag : BOOLEAN;

   BEGIN
```

```
-- rendezvous to get the information necessary for
-- initializing this device driver
ACCEPT set_up
       (su_ptr := IN ada_rtexec.configure_io.
                                 set_up_blk_access)

   DO

      pri_channel := su_ptr.ALL.primary_channel;

      -- register for external interrupt for this
      -- device driver task
      ADA_RTEXC.INTENTRY.DEFINE_DD_INT_ENTRY
         (which_entry =>
            ada_rtexec.configure_io.
                        external_interrupt_entry,
          which int    =>
 system.address
             (ada_rtexec.configure_io.
                          external_interrupt_monitor
              + pri_channel
             )
         );

     -- initialize the primary channel
     ADA_RTEXEC.CHANNEL_IO.DD_REQUEST
        (function_pos => io_operation'POS(initialize),
         channel_number => pri_channel
        );

 END SET_UP;

 LOOP  -- accept make_request

   -- rendezvous to accept the requested I/O operation and
   -- the channel on which this device is to run

   ACCEPT make_request
      (iorb2)ptr : IN ada_rtexec.configure_io.iorb2_access)
      DO

         -- perform the write operation
         ADA_RTEXEC.CHANNEL_IO.DD_REQUEST(
            function_pos    => io_operation'POS(write),
            channel_number  => pri_channel,
            transfer_count  => iorb2_ptr.ALL.iorb.
                               data_length,
            buffer_address  => iorb2P_ptr.ALL.
                               physical_data_location,
            command         => 0,
            ei_address      => ei_word'ADDRESS);
```

```
            -- return from ADA_RTEXEC.CHANNEL_IO
            SELECT
               ACCEPT  external_interrupt DO

                  ei_accept_flag := TRUE;

               END external_interrupt;

            OR  -- time out
               DELAY 3600.0;
               -- allow time to complete the write

               -- do not want the interrupt to occur later
               ADA_RTEXEC.CHANNEL_IO.RESET_CHANNEL(pri_channel);

               ei_flag := FALSE;

            END SELECT;

            IF ei_accept_flag THEN

               -- pass the external interrupt
               iorb2_ptr.ALL.iorb.ei_word := ei_word;

               -- set return status to operation successful
               iorb2_ptr.ALL.iorb.status := io_defs.ior_ok;

            ELSE

               -- set return status to operation unsuccessful
               iorb2_ptr.ALL.iorb.status := io_defs.
                                            ior_oper_err;

            END IF;

         END make_request;

      END LOOP;

   END FX286_DD;

END FX286_DEVICE;
```

Step (8) The Device Driver Channel/Chain Program

The channel/chain program for a peripheral device maintains the following format. A five word header for use by the Exporter, a channel

program jump table for CHANNEL_IO to enter, a data region, channel programs, and chain programs.

The channel/chain program employs an Ada importer directive file. This file (IMPUYK43) converts an assembly language compilation unit into a format required for input to the linker. The directives file provides entry point and reference information that ties the assembly code representation to the Ada package specification for which it was developed. The commands needed in the importer directives file for this example are the following:

```
.IOCHANNEL                                -- identifies a
                                          -- channel/chain program
.AC EXECUTION 0                           -- all code must be in
                                          -- the execution psect
.SUBPROGRAM FX286PAR FX_286_PARALLEL      -- identifies the
                                          -- subprogram
```

The program specification is as follows:

```
PROCEDURE FX_286_PARALLEL

   -- FX_268_PARALLEL contains the channel/chain program that
   -- will initiate the I/O chains and perform the requested I/O
   -- operation on a parallel interface to an EPSON FX_286
   -- printer.
   PRAGMA INTERFACE (MACRO_NORMAL, FX286_PARALLEL);
```

The program body is as follows:

```
*ULTRA FX286PAR/RTEXEC
       OPTIONS UYK43, SOURCE, OBJECT, EMBED, ACGEN

$(0)
     -INCLUDE INC.REGISTER
     - INCLUDE INC.IOC

    IOC$NO      EQU    0            . IOC number
    CHAN$NO     EQU    0            . channel number
    CPUSTRT     CPUSRT          . offset of first CPU instruction
    IOCSTRT     IOCSRT          . offset of first IOC instruction
    CHANNEL     RES 1           . bit encoded channel number
    CHAN        RES 1           . channel number
    IOC         RES 1           . IOC number

    FX286PAR*                   . channel program jump table
             J      WRITE,K0,B0,S5    . WRITE CHANNEL PROGRAM
                                      . send output buffer
```

```
         J       INIT,K0,B0,S5      . INITIALIZE CHANNEL
                                    . PROGRAM
                                    .  master clear the
                                    . printer
         . channel/chain data
CLEAR    00000000001                . enable printer EF

EIBCW    BCW     EIBUF,1            . external interrupt BCW
EIBUF    RES     1                  . external interrupt
                                    . buffer
OTBCW    BCW     0,1                . OUTPUT BUFFER CONTROL WD
OTXFER   RES     1                  . the output transfer
                                    . count
OTCAPBB  RES     1                  . the output CAP buffer
                                    . base
CPUSRT   . channel programs

WRITE
         . get output buffer physical address from input
         . parameters and save it

         LA      A1,3,K3,AP                 . put the buffer
                                            . address in A1

         . save the output buffer physical address for
         . loading into the output buffer base register
         . prior to the initiate output command
         SA      A1,OTCAPBB,K3,B0,S5

         . decrement and get the output transfer count from
          . input parameters
         RPD     2,K3,AP
         LA      A2,2,K3,AP
         SA      A2,OTXFER,K3,B0,S5

         . initiate I/O on this channel
         CHCL    IOC$NO,CHANNEL,B0,S5 . clear I/O channel
         IO      IOC$NO,WRITEC,B0,S5  . initiate the device
                                      . driver write chain
         . return
         JS      0,0,RA               . return to caller

INIT
         . initiate I/O on this channel
         CHCL  IOC$NO,CHANNEL,B0,S5   . clear I/O channel
         IO    IOC$NO,INITC,B0,S5     . initiate the device
                                      . drvr initialization
         . return
         JS      0,0,RA               . return to caller

         EJECT
```

```
IOCSRT    .CHAIN PROGRAMS:

WRITEC
          AOC      CHAN$NO,WRITECH,0     . activate output CAP
                                         . and release the CAR

INITC
          AFC      CHAN$NO,INITCH,0      . activate CAP
                                         . external function
                                         . and release the CAR

WRITECH
          AXC      CHAN$NO,EICH,1        . activate external
                                         . CAP interrupt
          ILB      OTXFER,K3,B0,1,S0     . output transfer
                                         . count is loop index
          ILS      OTCAPBB,K3,B0,1,S0    . load output CAP
                                         . buffer base
          ILA      1,K0,B0,1,S0          . load the accumulator
                                         . for incr. of 1
NEXTOT    OB       CHAN$NO,OTBCW,K3,1,0  . send data to printer
          IRA      OD$BCWB,*K2,B0,1,S0   . increment buffer base
          ICID     0,K0,B0,1,S0          . decrement loop index
          IJC      NEXTOT,B0,1           . jump if transfer is
                                         . NOT complete
          XMIR     CHAN$NO,0             . generate interrupt
                                         . to return to the
                                         . device driver
                                         . waiting task
                                         . terminate the chain
EICH
          XB       CHAN$NO,EIBCW,K3,1,0  . accept status of EI
          TFB      CHAN$NO,1,0           . terminate EF buffer
          TOB      CHAN$NO,1,0           . terminate the OUTPUT
                                         . buffer
          XMIR     CHAN$NO,0             . generate interrupt to
                                         . return to the device
                                         . waiting task
                                         . terminate the chain
INITCH
          FB       CHAN$NO,CLEAR,K0,0,0  . master clear printer
                                         . terminate the chain
          END
```

Step (10) Compiling and Linking the Device Driver

The RTE link container needs to be built to contain the new device driver specification and body. Prior to linking the executive, the new

device driver specification and body must be compiled into the RTE library. The commands to compile are

```
$ ADA43 fx286_device.ads /COMPILER_MAINT/FLAGS_CG2_G
```

The channel/chain program must be incorporated into the RTE library. The following command is needed to compile the specification of the FX-286 channel/chain program:

```
$ADA43 fx286_parallel.ads /COMPILER_MAINT/FLAGS_CG2_G
```

To import the body of the FX-286 channel chain program, type

```
$ common_defs := "directory name of DEF files"
$ IMP43 fx286_parallel.asm -
      /DIRECTIVES=fx286_parallel.idf -
      /EXTERNAL= 'common_defs
```

Step (10) Compiling and Linking the RTL

Compilation of the Run-Time Library is necessary when adding the new package in the Physical_IO layer, because subprogram FILE_IO.IO_REQUEST was modified. The new package PRINTER_IO must also be compiled:

```
$ADA43 file_io_lib.adb
$ADA43 printer_io.ads
$ADA43 printer_io.adb
```

Step (10) Configuring and Exporting the New Device

Two exporter commands are necessary to configure the new peripheral device:

```
SET VALID_DEVICE_MNEMONICS "PR","fx286_device"
SET CONFIGURED_DEVICE "PR"
```

```
' PRIMARY_CHANNEL => 1
' CHANNEL_TYPE    => "computer_device_32"
' MAX_UNITS       => 1
' ACCESS          => "sequential"
' CONTAINER_NAME  => "FX_286_PARALLEL.BODY"
```

4.3.2 Runtime Memory Layout

One of the most common modifications necessary to an embedded system is to ensure that it recognizes the amount of memory contained within the system. To do this, some of the operating parameters must be tuned. Careful optimization of these parameters, such as stack size, heap size, and main memory, will do much to allow particular applications to execute within the constraints of the system.

For example, an application, such as a communications switch, for which the entire set of runtime data structures and sizes are known, and for which additional runtime objects are not required, might be able to size the entire application into as little as 64K of memory by limiting (or eliminating entirely) the storage space allocated to the heap. This might allow a final application to run on a smaller embedded configuration.

In what follows the following implementations will be examined:

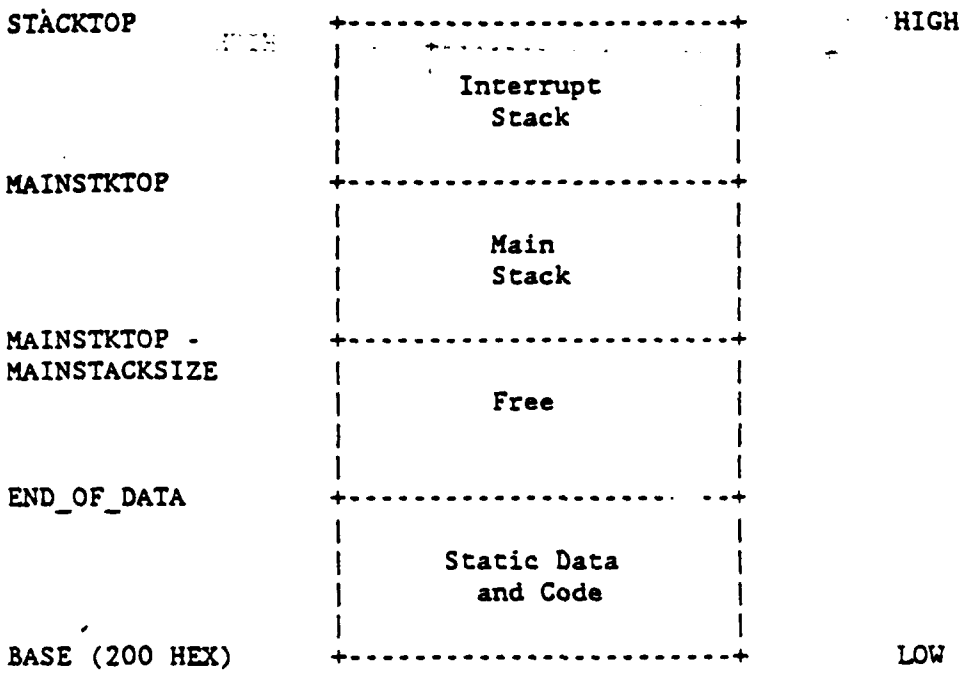
- Alsys
- Tartan
- TeleSoft
- Verdix

The Alsys Real Time Kernel (ARTE) for the PS/2 and PC AT -> 680X0 compiler locks for these specifications in the file usconxxx.asm (where xxx is target dependent). Some of these parameters are

```
VRTXCODE        VRTX code space low address,
VRTXWORK        VRTX workspace low address,
```

VRTXWKSZ	VRTX workspace size,
IOXCODE	IOX code space low address,
IOXWORK	IOX workspace low address,
IOXWKSZ	IOX workspace size,
FMXCODE	FMX code space low address,
FMXWORK	FMX workspace low address,
FMXWKSZ	FMX workspace size,
HEAPADR	low address of the workspace dedicated to ARTE,
HEAPSIZE	size of the heap workspace in bytes,
USTKSIZE	Ada task stack size (consisting of ARTE_STACK + ARTE_LOCAL_DATA_SIZE + USER_STACK_SIZE),
ISTKSIZE	interrupt stack size.

In the Tartan VMS -> 1750A compiler, this information is specified in the file *TADAHOME:USERDEFS.ASM*. The parameters provide a layout of memory as shown in the annotated display below:



The TeleSoft VMS -> 680X0 compiler provides for the specification of these values through the linker options files, such as VMEXXX.OPT (where XXX is CPU dependent). These linker directives include the following:

location and size of the main program's stack

```
DEFINE/ADA_USER_STACK_LOCATION=%X0D0000
DEFINE/ADA_USER_STACK_SIZE=%XC000,
```

location and size of the interrupt stack

```
DEFINE/ADA_INTERRUPT_STACK_LOCATION=%X0DE000
DEFINE/ADA_INTERRUPT_STACK_SIZE=%X2000,
```

location and size of the heap

```
DEFINE/ADA_HEAP_LOCATION=%x0DE000
DEFINE/ADA_HEAP_SIZE=%X08C000,
```

location and size of the master stack (MC68020 only)

```
DEFINE/ADA_MASTER_STACK_LOCATION=%XDC000
DEFINE/ADA_MASTER_STACK_SIZE=%X2000
```

In the Verdix VMS -> iAPX80386 compiler, these parameters are specified in the CONFIGURATION package in the file config.a. They include

KRN_STACK_BASE	the initial value of the stack pointer used by the debug monitor, from which the stack grows to low memory;
KRN_STACK_SIZE	the size of the kernel stack;
USER_STACK_BASE	the initial value of the user main stack, from which the stack grows to low memory;
USER_STACK_SIZE	the size of the user stack;
KRN_HEAP_BASE	the initial value of the kernel heap space, from which the heap grows to high memory;
KRN_HEAP_SIZE	the size of the kernel heap;
USER_HEAP_BASE	the start of the user heap space, from which the heap grows to high memory;
USER_HEAP_SIZE	the size of the user heap.

4.3.3 Clock Correlation

Another common modification is to ensure that there is a correct correlation between the RTE and the clock speed of the embedded processor.

In what follows the following implementations will be examined:

- o Alsys,
- o Verdix.

In the Alsys PS/2 & PC AT -> 680X0 compiler, this relationship is specified in the value for TICK in the configuration file:

TICK	hardware tick (in microseconds).

The Verdix VMS -> iAPX80386 compiler specifies COUNTS_PER_MSEC in the TIMER_SUPPORT package of the file timer_sup.a.

COUNTS_PER_MSEC	the number of times the time ticks per millisecond.

4.3.4 Processor Scheduling

A third, somewhat less common, modification relates to the type of scheduler for allocating the processor to the tasks ready to be run. Common options are to provide for a round robin scheduling algorithm (as opposed to a non-preemptive scheduling one) and to allow the specification of the time slice interval.

In what follows only the Verdix implementation will be examined.

In the Verdix VMS -> iAPX80386 compiler, these choices may be made with the values of TIME_SLICING_ENABLED and TIME_SLICE_INTERVAL in the CONFIGURATION package in the file config.a.

TIME_SLICING_ENABLED	if true, tasks will be preempted by time slicing; if false, a task will keep the processor until a delay or rendezvous is executed or an interrupt causes preemption;
TIME_SLICING_INTERVAL	length of the time slice interval.

5.0 WHAT ARE THE POSSIBLE SIDE EFFECTS OF TAILORING

Tailoring RTEs, is inherently risky. This section addresses the possible negative side effects of tailoring and ways to avoid them or to recover from them.

5.1 FACTORS TO BE AWARE OF

5.1.1 The Interface

Well defined interfaces between the application and the RTE, the compiler and the RTE, and among the modules of the RTE will minimize dependencies and provide an easily adaptable compilation system. When an Ada compilation system is customized, the internals of the system may become vulnerable to alterations if the method of communication between internal modules is not well defined. For example, choosing to interface through explicit parameters rather than by sharing access to global data structures will localize tailoring to specific modules and minimize the risk of side effects. The tedious and time consuming effort of tracking down the source of a "bug", which may have propagated from an "unrelated" tailored module, can be avoided if the compilation system maintains well defined interfaces.

As mentioned above, three levels of interface are worth noting:

- the interface that allows the application program to invoke RTE services explicitly,
- the interface by which the compiler explicitly requests services from the RTE,
- and the interface by which different components of the RTE communicate and request services from one another.

By understanding the interfaces within an RTE, it may be possible to localize customizations to a specific level of interface, thus decreasing the complexity and the risk of side effects.

5.1.2 Inline Code

During the execution of an application program, RTE actions may be initiated to support a specific language construct, or to provide a service that is solely in the domain of a traditional executive. If these services are implemented as code insertions by the compiler, rather than as subprograms, the tailoring effort will include modifications to the compiler. In this case not only does the tailoring effort require a greater level of expertise, but also the risk of side effects is increased. It may be difficult to predict how the inserted code (which has now been tailored) will affect the code in a module. Modifications to the compiler may also be required if calls to RTE subprograms are implemented as inline code; the compiler generated call must correspond with the tailored subprogram interface.

5.1.3 Hardware Considerations

Generally, a compilation system has been designed for a particular architecture. The compiler generates code in accordance with the machine's instruction set, and the RTE manages its resources. Two factors, then, which must also be considered when tailoring are the instruction set and the resources available.

The instruction set determines how specific Ada semantics are implemented and their dependence on the RTE. For example, if the instruction set architecture is capable of supporting arithmetic operations through generated code, there will not be a need for support routines. If the architecture for which the RTE is being tailored has a more limited instruction set, support routines will be needed. The latter, of course, will increase the probability of side effects because code is being called upon to emulate an operation that was previously trivial.

Resources that will affect the tailoring of an RTE are the processor speed, memory, and I/O capabilities. Most RTEs will be required to provide a specified amount of computation within required time intervals. Because

the consequences of missing a real-time deadline can vary from reduction of throughput to numerical inaccuracy to partial loss of system functionality, or even to total system collapse, the RTE may need tailoring in order to meet processing and performance requirements. While the time taken to perform system functions, such as process initiation, process termination, and context switching, is affected by the processor speed, the overall execution time is affected by the performance overhead of the support functions. Tailoring that affects the overhead of the following support functions will impact the overall performance of the system:

- o subprogram entry/exit,
- o parameter passing,
- o elaboration of a task object or a task type,
- o normal termination of a task object,
- o delay and timing services,
- o rendezvous,
- o the maintenance of an entry call in the queue and dispatching,
- o scheduling decision for tasks with explicit priorities,
- o examination time for attributes of task objects,
- o interrupt latency and execution time for response,
- o initialization time.

A limited amount of addressable memory may place restrictions on the size of the RTE and may require special paging and framing mechanisms for allocation. Such mechanisms will have a great impact on scoping and parameter passing; side effects due to visibility issues are associated with such mechanisms. A simple example of the complexity related to visibility occurs with the three types of references to objects: local, up-level, and global. A local reference is one for which the reference and the object referred to are within the same stack frame, up level references require a chainback mechanism to an enclosing activation record (it is more costly for each additional level separating the reference from the object), and global references (i.e., to objects with WITHed specs) refer directly to specific addresses embedded in the code by the linker. If the application program has been mapped into several frames (situations may exist where even a

subprogram unit may need to be mapped into several frames), for each one of these reference types and for cross frame calls, any parameters that are needed must be copied (to the stack) and a mapping to the new frame constructed; on return, the old frame is remapped, and any parameters that need to be copied back are copied. Framing also affects tasking; if a task resides in its own frame, each task switch will require the additional overhead of a remapping operation

Embedded computer systems tend to be input/output intensive; they are often interfaced with many devices that must handle different types of I/O. Each class of processor will usually have unique ways of handling I/O. Mechanisms to convert data from special purpose devices or to provide access to the bit and byte level may be needed to provide correct data representation to the RTE. In the case of low-level I/O, I/O ports and registers must be directly addressable to specialized devices. Side effects due to customizing the I/O mechanism will have less of an effect on other modules of the RTE because the I/O mechanism is somewhat autonomous; there are exceptions however. For example, the memory management scheme will be influenced by devices that are memory-mapped. These devices require that memory be updated in accordance with their byte, word, or long word transfers. Difficulties may arise when the SYSTEM.STORAGE_UNIT is a different size than the transfer unit. The unpleasant side effect is that the information sent in the previous transfer may be destroyed when, for efficiency reasons, the RTE updates its memory in multiple units.

5.2 RECOVERING FROM ERRORS INCURRED BY TAILORING

The tailoring of an RTE should be approached as any other Ada software development effort and must be fully supported by a system development environment. Typically, embedded software systems are developed and tested to the extent possible on the host and then downloaded and integrated with the target hardware for system level testing and eventual deployment. As with the tailoring of the RTE, those modules that will require modification

must go through the development process, the link with the rest the RTE, and the testing.

As part of the development effort, it is recommended that a full suite of tests be obtained or developed to provide, in accordance with accepted software development practices, unit- level, integration-level, and system-level testing. Before any customization begins, this library of tests must be executed on the target and the results recorded. In addition to the results of the tests, the original compilation system and its integrity must be maintained in order to establish a base line. This base line provides a method of identifying whether an error is an application error or an error resulting from tailoring the RTE; by executing on the base line the code that caused the error the source of the error can be more easily identified.

Another mechanism for recovering from an error is the use of a facility, such as a module manager or a library manager, where components can be developed separately and then combined for testing of the individual functions of the product. This configuration management approach allows for developing, maintaining, and testing versions of the components, and enables the tailoring team to recreate specific versions of the product. Software changes can be controlled and a record maintained of how the tailoring evolved and what its status was at any given time.
Ideally, the module manager will also be able to determine which modules in a system have been changed and which other modules are affected by the changes and be capable of building/rebuilding a system with variant version elements.

Tailoring an RTE will have an enormous effect on the integrity of an application program that was fielded prior to the customizations. It may be the case that new requirements for a particular system require that both the application and the RTE be customized. If it is only the RTE that is being customized, the magnitude of errors and side effects will depend on the level of customization; recovery from such conditions may be possible by modifying the application through "workaround" code until the problem is

solved. If, however, both the RTE and the application software are modified it will be difficult to determine the source of the error. As is the case with most development efforts, maintenance of the RTE may be required as problem areas are exposed through the test of time and use.

6.0 WHAT ARE THE COSTS OF TAILORING

6.1 COST IMPACTS OF TAILORING ON EMBEDDED SYSTEMS DEVELOPMENT

When an Ada RTE requires tailoring, the costs are imparted to development and maintenance of a system in three important ways:

6.1.1 Costs Associated with Responsibility

If an RTE is used in an Ada system exactly as delivered by the compiler vendor, the responsibility for errors in this software clearly lies with the vendor and it is his responsibility to fix them. If however the RTE is modified by the customer, the vendor can easily argue that he is absolved of responsibility. This is a good argument even if the error was delivered with the product, because a vendor should not be expected to sort out the customer's changes and errors that may result from those changes. The net effect of this transferral of responsibility is that the customer will need to spend the resources to localize a problem and prove the source of the fault.

6.1.2 Costs Associated With Expertise

To be able to tailor an RTE in-house requires the acquisition of low-level programming expertise. To be able to understand and correctly alter this system, a knowledge of the machine architecture and real-time programming is needed that is more specialized than that required for the application alone. Implanting such expertise in a development team entails costs for training existing employees or hiring RTE specialists.

6.1.3 Costs Associated with the Development Cycle

In traditional systems development it is assumed that any systems software that comes with the computer has been thoroughly tested and that enhancements will be supplied by the vendor. Once the RTE is tailored by the customer, these assumptions are no longer valid; and the RTE becomes an

additional software system, probably a complex one, that has to come under formal procedures.

Tailoring the RTE can impact the development cycle vertically by requiring that more, possibly specialized, staff be added to the team to perform RTE tailoring tasks. It can impact the development cycle horizontally by requiring that sequential testing and integration steps related to tailoring the RTE be inserted into the schedule. Specific impacts are as follows:

- Requirements Analysis/Planning - Requirements for tailoring the RTE need to be identified and planned for. Planning should include special tools if required. This requires that the scope of the tailoring and the commitment of personnel, time and materials be specified. For an experienced developer (both experience with a specific compiler and a specific type of application is needed), the need to tailor may be obvious. For a naive developer, this may not be recognized until the integration and testing phase when testing begins to uncover inadequacies in the RTE. In the latter case the impacts of tailoring will be considerably more severe.

- Software Design - The RTE and its associated RTL is comparable to a configuration item that has been obtained from another source but has to be customized for the current application. During the design phase vendor documentation and source code for the RTL need to be studied and locations of changes need to be pinpointed. A testing program for the RTE needs to be developed.

- Coding and Unit Testing - The changes to the RTL source need to be implemented. It is not clear how tailored RTE units might be tested in isolation; but perhaps, diagnostics can be developed that emulate higher level calls to these units. These would have to be designed during the design phase.

- Integration and Testing - Extensive testing of the tailored RTE will need to be done during this phase because the application is now being integrated and compiled for the target machine. To keep problems with the RTE separate from problems with the application, software diagnostics that are capable of testing the virtual machine in isolation from the application may be needed. Unfortunately for some developers, the need to tailor the RTE may first be recognized during this phase.

- Documentation and Configuration Control - Tailoring the RTE will further burden documentation activities and add an additional layer to configuration control.

- Field Testing and Maintenance - Costs incurred from tailoring may continue after installation. The cost of field testing and maintenance, as well as enhancements, will be increased due to the difficulty of separating RTE problems from application problems, the need to test and maintain the RTE, and the need to have additional software specialists on hand. Once the RTE has been tailored in house, it is unlikely that a compiler vendor will accept responsibility for poor performance.

6.2 AUTOMATION OF THE TAILORING PROCESS

As mentioned above, the RTE is another software system; and, if it needs to be tailored, many of the automated tools used to maintain application software can be applied to it. It is assumed here that tailoring will be confined to the RTL and that the source code for the RTL is available. If the RTL is largely written in a high-level language such as Ada, the job of tailoring and maintaining the RTE is just an extension of tailoring and maintaining the application program.

The first step in minimizing costs associated with tailoring the RTE and forming a sound footing on which to base automated tools is the recognition by the compiler system vendors that tailoring may be required, and the provision of RTEs that are highly tailorable. Of special importance are: comprehensive documentation on the RTE, RTS and RTL; practical advice on tailoring the RTE; easily available source for the RTL; and a modular RTE that has comprehensive functionality.

Weiderman, et al. (49) lists the following tools that may constitute an Ada embedded system development environment:

- Target-independent tools
  - Pretty printer
  - Language-sensitive editor
  - Static analyzer
  - Source code cross-referencer
  - Test manager
  - Configuration manager
  - Module manager
  - Library browser

- Target-dependent tools
  - Ada cross-compiler
  - Cross-assembler
  - Linker
  - System builder
  - Load module downloader/receiver
  - Symbolic, source-level debugger
  - Dynamic analyzer
  - Simulator
  - Real-time monitor

These tools generally apply to any embedded software system, and the functionality of some, like the cross-compiler, is self evident. However, several of these tools can be singled out as having particular relevance to the RTE. These are:

- The Test Manager - The test manager is used to organize, execute, and analyze software testing. In principle this is an important tool because one of the biggest issues related to tailoring the RTE is to provide testing that is capable of finding RTE errors. It appears unlikely, however, that there are currently any test managers that address RTEs directly.

- The configuration manager - This tool tracks generations and variants of an evolving software system. This is an extremely useful tool for tailoring an RTE for different systems or debugging and testing a tailored RTE. The RTE may be managed as a software system separate from the application. The RTE base line would be the RTE configuration supplied by the vendor.

- The module manager - The module manager works in conjunction with the configuration manager and tracks details about the evolution of individual modules, including module dependencies. This has obvious value if the tailorable RTE is sufficiently modularized.

- The cross-assembler - The cross-assembler works with the cross-compiler to enable assembled programs to be loaded into an Ada executable. This tool will be needed if the RTL source has been written in assembler or if the RTE has been tailored using assembler.

- The symbolic, source level debugger - A symbolic, source level debugger makes possible fine grained debugging of an Ada program on the target machine by providing selectable features such as breakpoints, traces and single-stepping. While such a debugger would be an obvious help in debugging a tailored RTE, existing

debuggers usually service the Ada application code and may not be able to service the RTE.

- The simulator - Of great importance to debugging RTEs are target simulators. These simulators execute on the host machine and emulate the functional and temporal behavior of the target machine. Such a simulator will permit the RTE/machine interface to be tested and debugged without the necessity of loading the target machine repeatedly. Ideally, the simulator would work in conjunction with a symbolic debugger that would be able to execute on the host machine and recognize the simulator as a real machine.

- The real-time monitor - A real-time monitor is hardware or software in the target machine that monitors timing and functional sequences. Because real-time performance of RTEs is a major issue, such a monitor would have great value in testing a tailored RTE.

The RTE, along with the bare machine, constitute a virtual machine for the Ada application (8). An additional tool, not described in Weiderman, et al., that would be of value in establishing that tailoring the RTE has not compromised the functionality or performance of the virtual machine would be one that generates diagnostics for the virtual machine. These diagnostics would test the RTE/application interface and also possibly work in conjunction with the simulator and symbolic debugger on the host machine. The diagnostics would allow the tailored RTE to be tested in isolation from the application software. Such diagnostic programs may be supplied by a vendor as part of the tailorable RTE package, along with instructions for their use.

Figure 5 illustrates where the use of these tools would occur in the software development cycle.

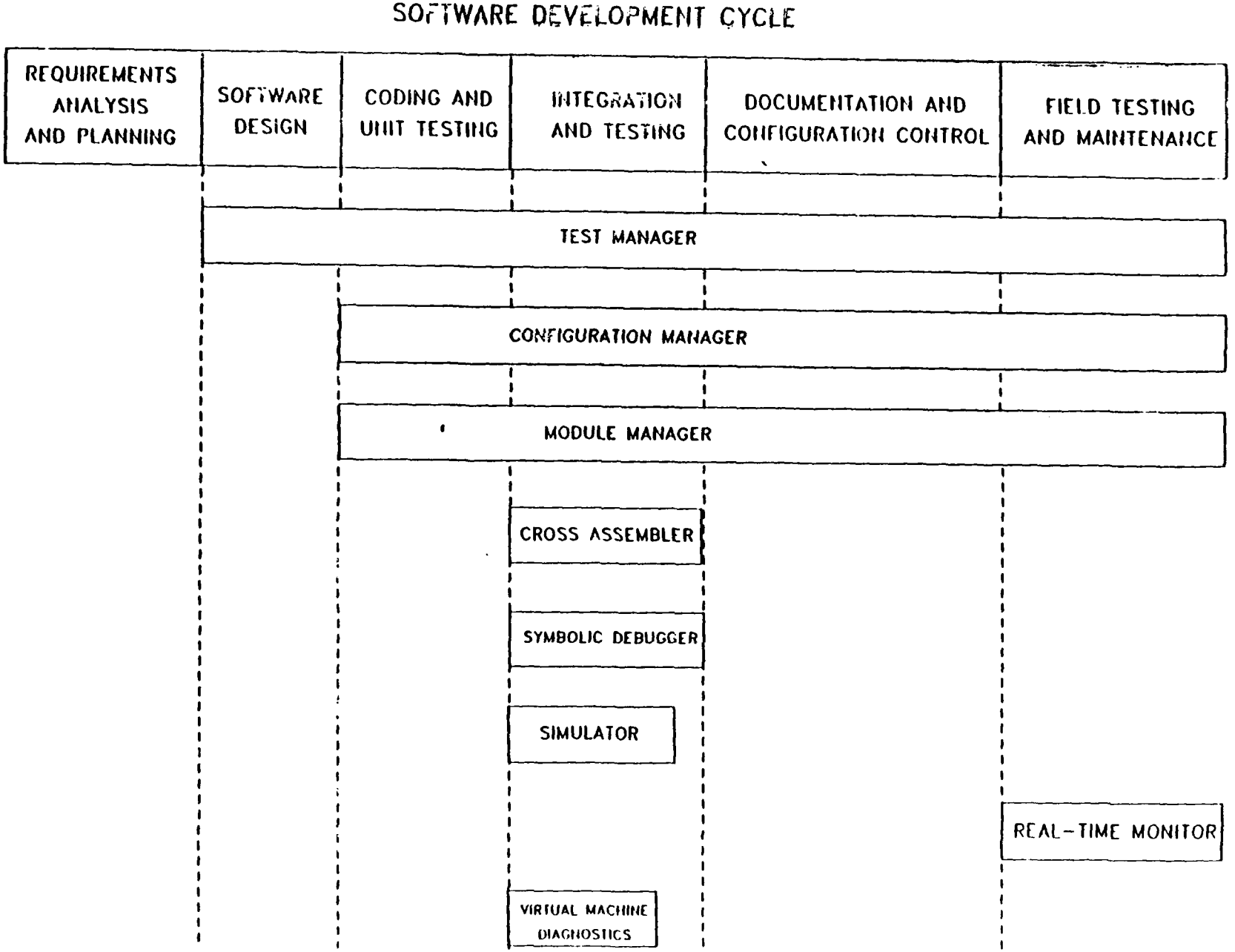


Figure 5. Application of RTE Development Tools

7.0 WHAT CAN VENDORS DO TO MAKE RTES MORE TAILORABLE

7.1 THE NEED FOR RTE TAILORABILITY

Vendors of Ada compilation systems generally recognize the need for tailoring the Ada RTE, and most provide support for specific applications. In fact, the request for tailoring support may be welcome because it offers vendors an opportunity to enhance the power of their product, possibly with the cost of testing and development partially borne by the customer.

Today, a customer often has to rely on the vendor or other external assistance because, even if the source code of the runtime library is made available, it is a formidable task for the customer's staff to learn the intricacies of a particular vendor's design and code for the RTE. Yet there are many reasons why dependence on vendors may need to be minimized. Most of these reasons fall into the following categories:

- Cost Control - An Ada user can better control costs with an in-house staff. There are several reasons for this. It is expected that the vendor needs to realize profits from a product whenever possible. Qualified vendor staff may need to be pulled off other vendor projects, and thus their unavailability may be costly to the vendor. Different customers may be competing for vendor services, resulting in a lack of timely availability. Negotiation with a vendor may be possible if it can be shown that tailoring for a specific application would enhance the capability of the vendor's compilation system.

- Liability - Questions of product liability would be more difficult to resolve if compiler vendors are asked to participate in an in-house development by modifying their own product. This may be somewhat lessened if the vendor only advises and does not make the actual changes, but the problem still exists.

- National Security - An Ada development may be a classified defense project. In this case a classification may cover such things as design features, performance goals and functional limitations that are impacted by the RTE. It might be very difficult to provide proper clearance or to properly isolate outside experts who may be asked to tailor the RTE.

- Corporate Security - An Ada system may be part of a product development where some degree of confidentiality must be maintained to retain a competitive edge. To invite the vendor to

tailor the RTE may open the development to the eyes of vendors employees who do not have a direct non-disclosure agreement with the customer. While a compiler vendor may not purposely divulge a customer's secrets, a wider dissemination of proprietary information always entails a greater risk of disclosure.

- Special Systems Design - The need to tailor an Ada RTE may be related to special systems development such as distributed processing, fault tolerance, parallel processing, etc. While vendor RTE specialists may be intimately familiar with the compilation system, they may have little understanding of the problem to be solved. It may be simpler in some cases for the in-house staff to learn the vendor's RTE than it is for vendor specialist to become cognizant of the special problem.

- Special Hardware - In a situation similar to that immediately above, a vendor specialist may have little understanding of special or experimental hardware, such as a missile warhead, that needs to be interfaced to.

- Lack of Availability - Vendor staff may simply not be available to the extent that is needed to tailor the RTE for an individual customer. This is aggravated by the possibility that the full extent of tailoring may not be recognized until integration and testing, and vendor assistance might be needed on short notice.

- Lack of Product Control - An Ada developer may have extensive software engineering practices in place to ensure a reliable and quality product. If a vendor or other external service is contracted to tailor an RTE, it may be more difficult to enforce these practices and to maintain the quality of the original Ada system.

In consideration of these reasons, vendors have to be encouraged not only to design RTEs that are highly tailorable, but also to provide information and tools with their products that enable customers to accomplish their own tailoring in a manner consistent with the Ada standard and established software engineering principles.

7.2 MODULARITY AND COMMONALITY AS A BASIS FOR TAILORABILITY

The tailorability criteria outlined in Section 3.0 and development tools described in Section 6.2 should provide guidance to vendors of compilation systems as to what is needed to make their RTEs more tailorable.

The key to this is the development of common, modular RTEs. Our reasons are as follows:

7.2.1 Modularity

A basic concept in Ada is the separation of packages, subprograms, and tasks into specification and body parts. To improve the tailorability of RTEs this theme of modularity should be extended to the RTE. The application of modularity to the Ada RTE has been discussed in Section 3.1.5. Advantages of a modular RTE are summarized as follows:

- Modular design is a recognized principle of software engineering - Modular design promotes qualities such as understandability, maintainability, and confirmability in software systems. The presence of these qualities in the RTE design enhances the ability to tailor the RTE in a timely and correct fashion.

- Modularity supports Ada tenets such as abstraction and information hiding - RTL routines can be specified in the manner of Ada packages or subprograms, even if not actually coded in Ada. If it is required that a specific module be tailored, it is necessary only to conform to the specifications of the module being tailored and to those with which it interfaces.

- Modularity makes it possible to accomplish tailoring in manageable chunks - Tailoring requires an understanding of the detailed design of the RTE code. If sufficient modularity is provided, the programmer needs only to recognize and understand the functionality of the target module.

- Modular design supports both configurability and tailorability of the RTE - A modular design also supports configurability by providing a clearly specified set of modules that can be selectively linked into the RTE. In many cases this would be adequate to achieve a sufficiently functional and efficient RTE. In other cases tailoring will still be required and, for the reasons above, modularity also improves tailorability. In this sense, a configurable RTE is a first line of defense, and a tailorable RTE is a second line of defense. Baker (12) states this in the following way: " Clearly, the division of a compilation system into modules with parameters and clearly defined interfaces can assist in both tailoring and configuring. If tailoring is required, it can help to localize changes to a single module, and if an adequate choice of parameters or alternate module versions are provided it can reduce changes to configuration choices."

- Modularity provides benefits for the compiler vendors -Modular design improves usability of their products, simplifies maintenance and upgrades, and makes possible selective privacy when certain components of their RTE are considered to be either proprietary or too critical to tamper with.

An executable Ada program consists of translated Ada code and the RTE. The RTE consists of common coding conventions and a RTS. The RTS includes an optional kernel and a tailorable, configurable part derived from a modular RTL. The kernel, if it exists, could be tailorable but, alternately, may be proprietary and neither configurable or tailorable. The RTS may be designed in a modular, hierarchical structure to achieve sufficient granularity. Top-level modules may be decomposed, to any level appropriate, into lower-level modules.

It is important to point out that a modular design in itself is not sufficient to promote tailorability. The modularity to be implemented must also exhibit qualities such as functional independence and cohesion that characterize good modular design.

7.2.2 Commonality

An RTE is part of an executable Ada program and, as such, basic Ada tenets such as reusability and portability should be extended to the RTE if it is to be tailorable. To accomplish this, some level of commonality should be applied to modular designs for RTEs. A common partitioning theme is a model for dividing an RTE into logical and manageable modules. Development of a common partitioning theme is an important component of tailoring and is expected to provide the benefits that follow:

- An industry-wide understanding of RTEs - Ada developers and their staffs would be able to learn a body of knowledge about RTE architecture and functionality that is applicable to any Ada compilation system.
- Less dependence on a single vendor - An Ada developer would be able to migrate from one Ada compilation system for purposes of hardware choice or performance with a considerably shorter learning curve.

- Developer staffs can specialize - When tailorability is required, in-house programmers can specialize in specific RTE components that are common to all compiler products rather than learning a total RTE design that is unique to a single vendor.

- Lessons learned have wider application - Because a similar problem can be more easily localized and probably would require a similar solution, experience gained in tailoring one RTE will have wider application to RTEs derived from other compilation systems.

Even if the RTL is written in assembly code, specifications for common modules can be defined in Ada code. This would help to commonize tailorability practices and perhaps lead to a standard specification for RTE modules.

To achieve a common, modular approach it is also necessary to recommend a logical theme for partitioning the RTE. The taxonomy of real-time features presented in Appendix A of this document is one candidate for such a partitioning theme. Another is "The *Catalog of Interface Features and Options for the Ada Runtime Environment*," assembled by the Ada Runtime Environment Working Group - Interface Subgroup (18). Whatever the theme that might be selected, it should serve to minimize coupling so modules can be tailored without significant side effects.

7.3 BENEFITS FOR VENDORS

To make possible improved tailorability of RTEs and common, systematic approaches to tailoring, vendors of Ada compilation systems have to be willing participants. While this may imply some level adherence to formal or informal industry practices and standards, vendors are also likely to benefit from these developments. Potential benefits for vendors are as follows:

- Standard Practices - Standards have proven to be beneficial to the industry as a whole. Industry standards may be imposed formally like those in the communications industry or, like IBM compatibility, occur by default; in either case, they have proven to be an impetus to the industry rather than an inhibitor.

- Wider Acceptance of Ada - Early compilation systems for bare machines have shown weaknesses in the RTEs they generate. Any improvements in the performance of Ada RTEs and the ability of users to influence this performance will widen the acceptance of Ada as a language for real-time, embedded systems.

- Opportunities for New Markets - A wider acceptance of Ada will provide additional customers for vendors of Ada compilation systems. This may include military customers who have been reluctant to begin use of Ada and commercial customers who have been waiting for Ada to first succeed in the military arena.

- Opportunities for New Products - Common tailoring practices may form a basis for new products. Examples are development tools that assist in tailoring RTEs and independently produced RTEs that are tailored for specific application areas.

- Simplified Product Development/Maintenance - If, for example, modular RTEs are partitioned according to some industry norm and common module specifications are used throughout the industry, code and documentation for these modules would be much more efficiently enhanced and maintained.

8.0 CONCLUSION

The major goals of this report have been the following:

- o to clarify the concept of tailoring (Section 2),
- o to provide guidance on how to tailor an RTE (Sections 3 and 4),
- o to indicate some of the side effects and costs of tailoring, and to suggest ways of minimizing them (Sections 5 through 7).

Tailoring was defined in Section 2 as making changes to the code of an RTE in order to improve performance, improve utilization of computing resources, or implement functionality not obtainable through a particular version of an Ada RTE. It was concluded that tailoring usually involves modifying the runtime library; and that such modification may take the form of changing code within the RTL, replacing one module in the RTL by another written by the user, and/or adding code to an RTL module to extend its functionality.

Before committing resources to tailoring an RTE, a user should ask to what extent it is tailorable in the first place. In Section 3 it was suggested that this question can be answered by considering both design features of the RTE and services provided by the vendor. Section 4 provides a technical discussion of tailoring opportunities within the RTE, rules for tailoring, and several examples.

Because tailoring is an inherently risky venture, possible side effects and costs of tailoring are discussed in Sections 5 and 6, respectively. Ways of (at best) avoiding these side effects and (at worst) recovering from them are also suggested. The report concludes (Section 7.0) with a survey of what vendors can do to make RTEs more tailorable. It was concluded that the most important contribution vendors can make is to provide common, modular RTEs.

Ideally, an RTE should be designed so that, to achieve the functionality required by a particular application, it can be modified by

simply substituting one vendor supplied module for another. In other words, in an ideal world, the user of an RTE need only configure it, not tailor it. But until such time as the complete configurability of RTEs becomes a reality (if ever), tailoring, with all its inherent difficulties and dangers, will remain the only viable alternative. It is hoped that this report will contribute to a better understanding of tailoring on a theoretical level, and, on a practical level, will provide guidance in exploiting its advantages while avoiding its risks.

9.0 BIBLIOGRAPHY

1. Ada Compiler System, TLD Systems Ltd, Torrance, California.

2. The Ada Execution Environment VMS/68000, Telesoft.

3. "Ada for Embedded Systems: Issues and Questions," Software Engineering Institute, December 1987.

4. "Ada Language Implementations," Ada Information Clearing House, Ada Joint Program Office, October 1987.

5. "Ada/L Program Support Environment Handbook - Preliminary," Naval Sea Systems Command, Control Data Corporation, Bloomington, Minnesota, June 1988.

6. "Ada/1750A Problems and Solutions," Tartan Laboratories, Inc., Pittsburgh, Pennsylvania.

7. Ada 1750 Cross Development System Preliminary, Interact Corporation, New York, New York.

8. "An Ada Virtual Machine for Embedded Real-Time Applications," Computer Sciences Corporation, Final Technical Report to Center for Software Engineering, CECOM, December 1988.

9. Aho, A. and Ullman, J., Principles of Compiler Design, Addison-Wesley Publishing Company, 1977.

10. Allen, D., "Tailored Run-Time Environments for Real-Time Applications," Proceedings of the International Workshop on Real-Time Ada Issues, Ada Language UK LTD and ACM Press, May 1987.

11. Arthur, L., Measuring Programmer Productivity and Software Quality, John Wiley & Sons, New York, 1985

12. Baker, T., "Ada Runtime Support Environments to Better Support Real-Time Systems," Proceedings of the International Workshop on Real-Time Ada Issues, Ada Language UK LTD and ACM Press, May 1987.

13. Baker, T., "Issues Involved in Developing Real-Time Ada Systems," Final Technical Report to Center for Software Engineering, CECOM, July 1988.

14. Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.

15. Booch, G., Software Engineering with Ada Second Edition, Benjamin/Cummings, Menlo Park, California, 1987.

16. Bowen, Thomas P., Wigle, Gary B., and Tsai, Jay T., "Specification of Software Quality Attributes," Boeing Aerospace Company, Final Technical Report to RADC, February 1985.

17. Burns, A. and Wellings, A.J., "Real-Time Ada Issues." Proceedings of the International Workshop on Real-Time Ada Issues, Ada Language UK LTD and ACM Press, May 1987.

18. "A Catalog of Interface Features and Options for the Ada Run Time Environment Release 2.0," Ada Run Time Environment Working Group Interfaces Subgroup, Association for Computing Machinery Special Interest Group for Ada, December 1987.

19. "Catalogue of Ada Runtime Implementation Dependencies," Ada Run Time Environment Working Group Interfaces Subgroup, Association for Computing Machinery Special Interest Group for Ada, December 1987.

20. Configuring the Run Time VAX ULTRIX Self-Target, Verdix Corporation.

21. Cornhill, Dennis, "Limitations of Ada for Real-Time Scheduling," Proceedings of the International Workshop on Real-Time Ada Issues, Ada Language UK LTD and ACM Press, May 1987.

22. DDC-I Ada Compiler System Run-Time System Configuration Guide for DACS-80x86, DDC-I, Inc., Phoenix, Arizona, November 1987.

23. DDC-I Ada Compiler System User's Guide for DACS-80x86, DDC-I, Inc., Phoenix, Arizona, January 1988.

24. Draft Military Standard Defense System Software Development, DOD-STD-2167A, Department of Defense, Washington, D.C., October 1987.

25. "A Framework for Describing Ada Runtime Environments," Ada Run Time Environment Working Group Interfaces Subgroup, Association for Computing Machinery Special Interest Group for Ada, October 1987.

26. Fuhrer, J., et al., "E-MARS-Embedded Multi-processing Ada Runtime Support," Proceedings of the Sixth National Conference on Ada, March 1988.

27. Gries, D., Compiler Construction for Digital Computers, John Wiley and Sons, Inc., New York, New York, 1971.

28. Griest, L., Personal Correspondence, Labtek Corporation, May 1988.

29. Hsiao, D., Systems Programming: Concepts of Operating and Data Base Systems, Addison-Wesley Publishing Co., 1977.

30. IEEE Standard Glossary of Software Engineering Terminology, The Institute of Electrical and Electronics Engineers, Inc., New York, New York, February 1983.

31. "Interim Software Data Collection Forms Development," IIT Research Institute, Final Technical Report to RADC, June 1985.

32. Knight, J.C., "Session Summary Run-Time System Issues," Proceedings of the International Workshop on Real-Time Ada Issues, Ada Language UK LTD and ACM Press, May 1987.

33. Landwehr, R. and Hensel, P., "A Model for a Portable Ada Run-Time Library," Proceedings of the International Workshop on Real-Time Ada Issues, Ada Language UK LTD and ACM Press, May 1987.

34. "A Model Runtime Environment Interface for Ada," Model Runtime System Interface Task Force, Association for Computing Machinery Special Interest Group for Ada, January 1988.

35. Myers, G., The Art of Software Testing, John Wiley and Sons, New York, 1979.

36. Parnas, D., Clements, P. and Weiss, D., "The Modular Structure of Complex Systems," Transactions on Software Engineering, Vol. SE-11, No. 3, March 1985.

37. "Preliminary Guideline to Select, Use and Configure an Ada Runtime Environment," LabTek Corporation, Interim Technical Report to Center for Software Engineering, CECOM, March 1988.

38. Presson, P. et al., "Software Interoperability and Reusability Guidebook for Software Quality Measurement," Boeing Aerospace Company, Rome Air Development Center, RADC-TR-83-174, Vol. II, Rome, New York, July 1983.

39. Randall, C. and Rogers, P., "Distributed Ada: Extending the Runtime Environment for the Space Station Program," Proceedings of the Sixth National Conference on Ada, March 1988.

40. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983, United States Department of Defense, Washington D.C., February 1983.

41. Runtime Libraries VMS/1750A, Tartan Laboratories.

42. "Software Engineering Issues on Ada: Technology Insertion for Real-time Embedded Systems," LabTek Corporation, Final Technical Report to Center for Software Engineering, CECOM, July 1987.

43. "Software Engineering Problems Using Ada in Computers Integral to Weapons Systems," Sonicraft Corporation, Final Technical Report to Center for Software Engineering, CECOM, July 1987.

44. Seymour, B., "Bare Machine Ada Solves Real-Time Problems," Computer Design, Vol. 27, No. 5, March 1, 1988.

45. Wilson, R., "Embedded Systems Manipulate Distributed Tasks," Computer Design, Vol. 26, No. 16, pp. 49-61, September 1987.

46. VAX Ada Language Reference Manual, Digital Equipment Corporation, Maynard, Massachusetts, February 1985.

47. VAX Ada Programmer's Run-Time Reference Manual, Digital Equipment Corporation, Maynard, Masschusetts, February 1985.

48. VAX/VMS Host to MC680x0 Target Compiler User's Guide, Telesoft, San Diego, California, November 1987.

49. Weidermun, N., "Ada for Embedded Systems: Issues and Questions," Software Engineering Institute, Pittsburgh, Pennsylvania, 1987.

50. Wulf, W., et al., "The Design of an Optimized Compiler", Elsevier Nort-Holland, Inc., New York, New York, 1975.

APPENDIX A
TECHNICAL DISCUSSION OF THE RTE

The Ada standard views library units (library units include main programs) as being at the outermost level in the hierarchy of program units. The RTE, however, only recognizes tasks, as the LRM (10.1 (8)) specifies that each main program is treated as if it is called by some environment task. The compilation system is therefore responsible for translating an Ada program or collection of library units into code that will have the same form as that generated by a task type. The RTE will treat these as any other Ada task. This section describes the interface between the code generated by the Ada compiler and the runtime environment.

For each Ada compilation unit a compiler produces a translation of that unit into machine language. The collection of machine code produced is called the translated image of the source program. This image must be linked in with components of the RTE to produce a complete executable. The RTE consists of code that is invoked implicitly by the compiler-generated machine code, and explicitly via WITH clauses and subprogram calls. This implicit portion of the RTE is the RTL (see Figure 1). It is the RTL that is called upon to provide the traditional "operating system" functions such as task creation, termination, interrupt response, etc., which will likely require tailoring for real-time embedded applications.

For the purposes of this discussion, the RTE is divided into the following parts:

- task identification,
- task dependence,
- task activation,
- task scheduling,
- task synchronization,
- task termination,
- priorities,
- interrupts,

- o timing services,
- o storage allocation,
- o exceptions,
- o input/output.

This taxonomy was influenced by, but is not identical with, that presented in the MRTSI specification (34).

A.1 TASK IDENTIFICATION

The RTE identifies an elaborated task object by means of a unique identifier. Task identifiers are implemented as a type. One value of this type is used in situations where an identifier that does not correspond to any task is needed; it is used to identify what is referred to as the NULL task. In some cases (e.g., distributed or secure implementations) there may be more than one valid ID for a given task.

The ID of a task that is directly dependent on a master is only valid for use according to the Ada scoping rules. Task IDs may be represented in fixed-length format of 8 to 32 bits. The task ID may be implemented as the address of the task control block that is used by the RTE to store information associated with the task, or as an index into a task table.

Data structures containing task information are maintained and accessed by the RTE. The compiler provides the RTE with the information it needs to build these data structures. One of these items of information is the task ID. It is built by the compiler and initialized by the generated code. Given a task ID, the RTE is able to generate, access, and modify data structures (e.g., the Task Information Block, the Task Control Block, the Task Object Record, the Task Stack) associated with the corresponding task. These data structures form the interfaces between the compiler, the generated code, and the runtime library. The application program, however, is not able to access the task ID. It is strictly for use by the RTE.

A.2 TASK DEPENDENCE

Each task depends on at least one master. A master is a construct that is either a task, a currently executing block statement, a subprogram, or a library package. A task can be directly dependent on a master in two ways: (1) if it is activated by allocating an access object, it depends on the construct in which the corresponding access type is declared; (2) otherwise, it depends on the construct in which it itself is declared. A task is indirectly dependent on a master if it is activated by a master on which it is not directly dependent. (LRM 9.4 (1 - 4))

In the case where there is dependence on a master, the RTE must be notified of a change in the master nesting when execution is about to enter or leave a master. A master ID implementation is responsible for keeping track of the dynamic nesting of masters. This master ID may include path names or encrypted records as information data structures. The compiled code must be able to obtain the ID of the current master for use in creating a task. The compiler is responsible for keeping track of which program units are task masters and their nesting in the program text. The maintenance of all data structures necessary for keeping track of the currently executing master and the dependency relations of tasks on masters is accomplished by the RTE.

Actions that the RTE must perform before exiting from a master include termination of any dependent tasks that have not been activated (i.e., when an elaboration which creates the task raises an exception prior to its activation), termination of activated tasks dependent on the current master, deallocation of storage and task IDs associated with these tasks, and updating the RTE's external record of the currently executing master's ID.

A.3 TASK ACTIVATION

The creation of the task occurs as part of the elaboration of an object or the evaluation of an allocator. When a task is to be created, several

RTE functions take place as it progresses through its activation, normal execution, completion, and termination:

- The storage size and static priority for the task is specified.
- The master on which this task depends is identified.
- The number of entries of the task are determined.
- A unique ID for the task is assigned.
- All the data structures that will be needed to handle other RTE calls needing the ID as a parameter are initialized.
- The initial state of the new task is saved for the time when it begins activation (e.g., machine specific initial values for certain registers, the entry-point address of the activation code, information needed to initialize its workspace).

The method of handling storage allocation will depend on machine- specific conventions and must be established between the RTE and the compiler.

A task that is created by the evaluation of an allocator is activated after any initialization of the object created by the allocator. It is the responsibility of the compiler to ensure that the bodies of the tasks to be activated have been elaborated.

The structure of the code for a task will include the code to initialize its workspace, code for elaboration of its local declarations, calls to the RTE to signal that the task has completed activation, (so it can release its creator), code for the sequence of statements of the task body, and calls to RTE procedures that will perform actions required on completion of a task.

A.4 TASK SCHEDULING

The scheduling requirements outlined in section 9.8 of the LRM come into play when there are more tasks eligible for execution than can be supported simultaneously on a resource. Ada scheduling rules are based on a

priority scheme. The priority of a task may be set with the compile time pragma PRIORITY. This priority is not subject to change except as defined below with reference to a rendezvous.

The rules followed by the RTE in scheduling tasks with explicit priorities include:

- If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same processing resources, then the task with the higher priority must be allowed to execute. There is considerable implementation freedom to define when scheduling is to take place and what criteria are to be used in matching a task to a processor.
- For tasks of the same priority, the scheduling order is not defined by the language and the implementation may select the scheduling strategy.
- In priority based scheduling, the following rules apply where two tasks are engaged in a rendezvous: if both have explicit priorities, the rendezvous is executed with the higher of the two priorities; if only one of the two priorities is defined, the rendezvous is executed with at least that priority: if neither of the priorities is defined, the priority of the rendezvous is undefined.
- If a task is not explicitly given a priority, the implementation is free to choose an assumed priority.

A.5 TASK SYNCHRONIZATION

Entry calls and accept statements are the primary means of synchronization of tasks and of communicating values between tasks. The number of entries of each task is established at the time of creation. Every entry may be viewed as a simple entry or as a component of an entry family. The number of entries of each task is supplied to the RTE. The transmission of parameters between tasks can occur in two ways--via the runtime stack or by means of a pointer. The details of data representation are handled by the compiler. It is the RTE's responsibility to pass to an accept statement the information needed for addressing the parameters. The types of rendezvous that can be performed and the types of parameters that

can be passed may be limited by the amount of task storage space available for the copying of parameters to the accepting task's stack.

The three forms of entry calls that must be implemented are simple, timed, and conditional. The RTE is responsible for checking that the called task is callable and raising TASKING_ERROR if it is not. If the called task is not ready to rendezvous on a conditional call, control must be returned to the calling task immediately; for a timed call, a wake-up event (delay count down) must be scheduled; for a task that is not ready to rendezvous, the calling task must be blocked. Whenever a task entry is accepted, the parameters must be transferred at the beginning of the rendezvous.

The accepting of an entry call may be a simple accept statement or a selective wait statement with only accept alternatives, with delay alternatives, with an else clause, or with a terminate alternative (LRM 9.7.1(3)). One RTE routine may be called to implement all the forms of the selective wait statement. The compiler can translate the selective wait statement as a CASE statement; the code must evaluate any guards of the selective wait.

The LRM does not specify the algorithm to be used for selecting among open accept statements within a selective wait statement. Usually, an open accept statement will be checked in the order in which it appears in the selective wait statement. As soon as an open accept statement is found for which a corresponding entry call is pending, the accept statement is selected and the rendezvous initiated. If no rendezvous is immediately possible for any of the open accept statements, the selection algorithm must proceed as required in Section 9.7.1 of the LRM.

The evaluation of guard conditions is also implementation dependent. Usually, guard conditions are evaluated in the order in which they occur in the selective wait statement. The compiler-generated code evaluates all alternatives, including delay alternatives, and provides this information to the runtime system, which determines the appropriate action to take.

Delay alternatives are selected if and only if no rendezvous is immediately possible and the specified delay expires before a rendezvous is initiated. A delay expiration (whether established in conjunction with a delay statement, a selective wait statement, or a timed entry call) causes the delayed task to become eligible for execution immediately.

For accepting an entry call the actions of the RTE include raising the acceptor's dispatching priority and returning control to the creator indicating the selection of the accept alternative and the location of the parameter block, if any. If no call can be accepted and if the mode is not the ELSE mode, the RTE blocks the accepting task and records that it is waiting in a selective wait; for the ELSE mode, the RTE causes an immediate return if there is no call that can be accepted; for the DELAY mode, the RTE schedules a wake-up; for the TERMINATE mode the RTE checks for possible termination of all dependent tasks of a master.

To end the rendezvous, the RTE may have to lower the dispatching priority of the accepting task, transmit entry parameters back to the caller, propagate an exception, and then finally release the caller.

A.6 TASK TERMINATION

The rules for termination center around the dependencies among individual tasks and the readiness of related tasks to terminate. A task is dependent on its master program unit. A program unit's dependents tasks and the succession of tasks that may be descended from them establish dependencies that must be recorded and monitored by the RTE. Because to these dependencies, the final status of an individual task's execution affects the execution and termination of its dependent and master tasks.

The principal rule of termination prescribes that any program unit whose execution has completed cannot be left until all its dependent tasks (if any) either have terminated or have reached a juncture where they are eligible to terminate (namely, the task has completed or it is at an open terminate alternative).

The RTE must assess terminate conditions when:

- o A block or subprogram with dependent task has completed. Dependent tasks must be checked to see if they have terminated or are waiting to terminate. If so, those tasks that are waiting (if any) are terminated, and the subprogram is left. Otherwise, the thread of execution associated with the subprogram is suspended until the condition is met.

- o A task completes. Its dependents must be checked to see if all of them have terminated or are eligible for termination. If so, the waiting tasks (if any) and the task are terminated. Otherwise, the task is suspended until the condition is met. When the task is terminated, the RTE checks to see if it has a master waiting on it.

- o A task reaches a terminate alternative in a selective wait statement. A termination check, in accordance with the semantics of a selective wait, with terminate is performed.

Actions performed by the RTE upon task termination include checking the pending calls of the task for possible TASKING_ERROR in calling tasks, ensuring that the attributes CALLABLE and TERMINATED of a task evaluate as FALSE and TRUE respectively, and optionally, deallocating the working storage of the task.

A.7 PRIORITIES

If two tasks with different priorities are both eligible for execution, the one with the higher priority must be dispatched. Even though a task's inherent priority is static, its priority can be raised when it is engaged in a rendezvous with a higher priority task. The RTE itself maintains the dispatching priority of the task. The RTE interface must provide a way for the compiled code to make the static priority of a task known to the RTE at the time it is created.

The LRM defines PRIORITY to be a subtype of INTEGER whose range is implementation defined. The internal priorities used by a runtime system implementation may have wider range or finer granularity than those that are

supported by an implementation interface. Such priorities may be used to distinguish tasks handling interrupts, or to reflect differences between tasks declared to have equal standard priorities. Priority must also apply to all processing resources that the system must have (e.g., buses, process memory, and peripherals).

A.8 INTERRUPTS

The management of interrupts by a runtime environment is highly implementation dependent. Interrupts can be handled in the traditional way, by loading the address of the interrupt routine into a hardware interrupt vector, or by mapping the interrupt to a task (called an interrupt task) for direct or indirect handling of interrupts. The former method is usually implemented for interrupts that are critical to the implementation and for which strict control must be maintained. The latter allows for flexibility in handling interrupts; the interrupt acts as an entry call issued by a hardware task whose priority is higher than the main program. In either case, extreme care must be taken when tailoring to avoid corrupting RTE internals. In this section we are primarily concerned with the transfer of control to the accept statement of an interrupt task.

When handling interrupts directly, the accept body bound to an interrupt entry plays the same role as the traditional interrupt service routine: as soon as the interrupt occurs, control is transferred to the interrupt service routine, and this routine executes to completion. When interrupts are to be handled indirectly, that is, as a signal that an interrupt has occurred, the accepting task is notified of the event, but there is no hard deadline for reacting to this notification. The RTE associates the entry of a given task with specified interrupt, and every occurrence of this interrupt is then transformed into an entry call. If the called task is waiting at a matching accept statement, a simple rendezvous occurs; otherwise, the RTE will either treat the interrupt as a conditional entry call or record the occurrence in an internal data structure that it is waiting at a selective wait.

As previously mentioned in Section 9, the execution of all Ada code is viewed as taking place under the control of some environment task. The RTE maintains information on which task is currently executing and which tasks are eligible to execute. Any transfer of control due to an interrupt will involve the reading and updating of various tasking data structures in the RTE. Associated control information is passed to the interrupt entry as one or more parameters. The form of this information depends on the nature of the specific interrupt and the hardware that generates it. The RTE assumes the role of the calling task, and transfers the needed information to a location where it can be accessed via the normal parameter mechanism for entry calls. In the case where an interrupt entry call is queued, the parameter information is stored until the rendezvous is complete. The representation and layout of the entry parameters are determined by the RTE.

In the simplest case, the handler bound to an interrupt is the compiler-generated code for the accept body of the corresponding interrupt entry. This code includes a sequence of statements to save and restore the state of the interrupted computation and hardware and any parameter block information. The RTE must insure that the task executing the handler cannot be interrupted until it reaches the accept statement. During the execution of a handler, the internal data structures of the RTE are temporarily out of date. This happens because any operation leading to a wait state of the current task may cause the RTE's data structures to indicate that the current task was the last task that executed prior to the transfer of control to the handler, and not the task that owns the entry for which the handler provides the accept body. At the end of the interrupt rendezvous, the RTE cancels all changes (resets) that were affected by the hardware when the interrupt occurred.

A restriction that may apply to interrupt tasks is that an interrupt task may have only one entry, the interrupt entry. Implementations may also restrict the sequence of code statements inside the accept statement such that it cannot contain or invoke code that: (1) changes the masking state of the hardware, (2) references types or objects declared in scopes that enclose the interrupt task, (3) will cause the interrupt rendezvous to

occupy more than the given amount of stack space, or (4) may lead to the raising of an Ada exception.

A.9 TIMING SERVICES

The timing services of an RTE provide an application with the ability to specify absolute time or time intervals and to control the execution of a program in a time- sensitive manner. These services are provided by the predefined package CALENDAR and by the Ada delay statement.

Absolute time is represented by values of the Ada type CALENDAR.TIME, while time intervals are represented by values of type DURATION. The library package CALENDAR provides various operations that can be performed using values of these types. An implementation of package CALENDAR must be a part of any Ada runtime system and can easily be written in standard machine- independent Ada. A CALENDAR.TIME value is like, but does not correspond exactly to, real-world time. It is initialized from a reference year, month, day, and part of day by the runtime system. Because the representation of time needed for CALENDAR.TIME values differs from values of time kept by a machine's real-time clock, the package CALENDAR must perform all the necessary conversions. The real-time clock is initialized exactly once to zero at system initialization time. The value of the real-time clock cannot be changed during execution.

Control of program execution is accomplished by suspending processing of a given task for certain periods of time. The Ada standard specifies that it shall be possible for a task to delay its own execution for a guaranteed minimum amount of time. The amount of the delay is specified as a value of type DURATION. It is not possible, therefore for a task to delay itself for a time interval with greater precision than that afforded by the type DURATION; and that accuracy depends on the runtime system implementation. The implementation of the delay is the responsibility of the RTE and should be supported for the full range of type DURATION (which must extend to at least one day).

Implementation choices of the delay statement include tradeoffs between accuracy versus execution overhead, range of delay duration versus accuracy and execution costs, and execution costs versus the direct cost of setting up delays and (in the case of the select statement) canceling them. Its implementation strategies are likely to vary from one application to another. The accuracy of the delay will depend on the runtime system and implementation and also vary according to the system load.

A.10 STORAGE ALLOCATION

Storage is required for the application and the runtime system. While allocation of storage for the code is handled prior to runtime (i.e., by the linker and loader), the dynamic allocation of data storage is the responsibility of the runtime environment. The compiler-generated code requests block storage from the RTE and allocates this storage, which may or may not be a continuous block of addresses, to the appropriate task. The amount of workspace allocated to a task is determined by the compiler as well as by structures such as activation records and stack frames. The address of the first instruction of the activation code for each task must be provided to the RTE by the time of task activation.

Some variation of the following data structures will be implemented by a compiler system to provide the interface between the runtime system and the compiler:

- a task descriptor built by the compiler and initialized by the generated code to describe a single task or task type to the RTE,
- a task information block built by the compiler and initialized by the generated code to list the dependent tasks, blocks, or subprograms, and to note how many of those tasks are still active,
- a task object record created by the runtime system and initialized during task activation to keep track of the activation state of the task (namely, active, aborted, completed, terminated, or ready to terminate),
- a task control block to hold the information necessary to resume execution of the task and to effect a rendezvous with another task,

- a task stack, which is allocated when the task control block is created.

When a task is created, the runtime system returns the physical address of the task object record to the generated code and then passes this address to the RTE for all references to the task.

When there are no user-defined tasks, there is in effect only one task--the main program. In this case the program requires only a single heap for all allocation and a single stack for all procedure or function calls. The stack grows and shrinks as each subprogram is called, executes, and completes. With tasks, each task needs its own stack. Any subprogram called by a task, as well as subsequent subprograms called by that subprogram, will make use of the stack of the calling task.

There may be variations among implementations as to allocation of unconstrained arrays and access types. These may be allocated from the heap rather than the task stack, and in separate collection areas.

A.11 EXCEPTIONS

The exception processing mechanism supports the transfer of control from the normal flow of execution within a program to a handler as a result of an error or an exceptional situation. The transfer of control may be to a local handler or it may involve an upward scope propagation until a handler is found. Handlers may be tracked dynamically or mapped statically. The exception processing mechanism caters to exceptions caused by an operation or a direct request via the raise statement. It provides a level of reliability and fault tolerance to the RTE. The language defined CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, TASKING_ERROR, the "catch all" OTHERS, and additional implementation defined exceptions are supported by this module. Tasking Support and Memory Management will usually be invoked when propagating an exception.

A.12 INPUT/OUTPUT

In implementing I/O the language defined I/O operations are mapped to those of the target system's support utilities by means of device drivers. These device drivers will utilize common low-level interface routines and distinguish between terminal files and mass storage files by interfacing with some form of generic packages such as terminal I/O and tape I/O. The two categories of input/output to be implemented are binary and text; these can be broken down to sequential, direct, text, and low-level I/O. Sequential and direct I/O provide binary, sequential and random external file accessing. Text I/O provides human readable input/output to devices such as a terminal or printer. Low level I/O is likely to be the main form used by embedded systems. The SEND_CONTROL and RECEIVE_CONTROL primitives allow the application program device drivers to interact directly with the hardware; the implementation must assure this interface.